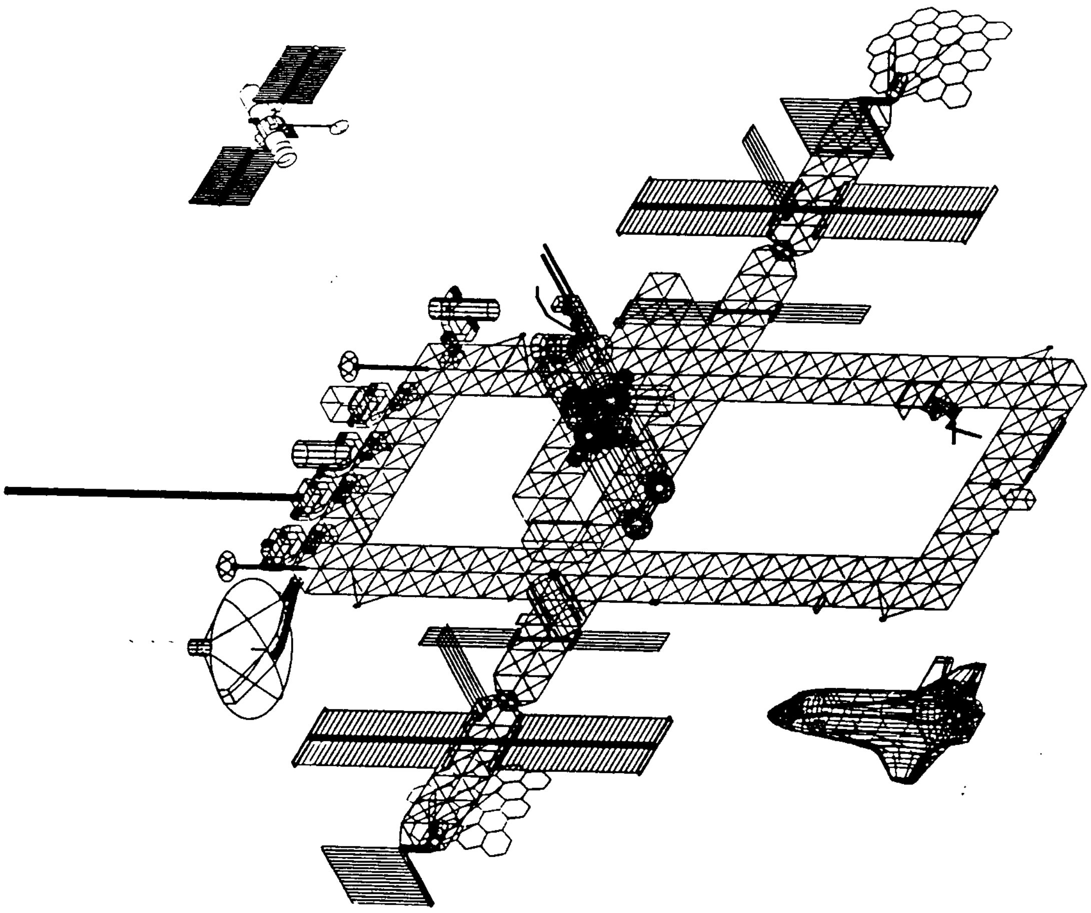
VOLUME II

First International Conference on Ada® Programming Language Applications For The NASA Space Station



June 2 - 5, 1986

Hosted by:

University of Houston-Clear Lake
School of Sciences and Technologies
High Technologies Laboratory

NASA Lyndon B. Johnson Space Center

In Cooperation with Local Contractors

(NASA-TM-101202) FIRST INTERNATIONAL CONFERENCE ON Ada (R) PROGRAMMING LANGUAGE APPLICATIONS FOR THE NASA SPACE STATION, VOLUME 2 (NASA) 504 p CSCL 09B

N89-16326
--THRU--
N89-16376
Unclas
G3/61 0167071

N89-16327

Integrating Automated Structured Analysis and Design with Ada Programming Support Environments

Alan Hecht and Andy Simmons
Cadre Technologies Inc.
222 Richmond St.
Providence, R.I. 02903
(401) 351-5950

Abstract

Ada Programming Support Environments (APSE) include many powerful tools that address the implementation of Ada code. These tools do not address the entire software development process. Structured analysis is a methodology that addresses the creation of complete and accurate system specifications. Structured design takes a specification and derives a plan to decompose the system sub-components, and provides heuristics to optimize the software design to minimize errors and maintenance. It can also promote the creation of reusable modules. Studies have shown that most software errors result from poor system specifications, and that these errors also become more expensive to fix as the development process continues. Structured analysis and design help to uncover errors in the early stages of development. APSE tools help insure that the code produced is correct, and aid in finding obscure coding errors. However, they do not have the capability to detect errors in specifications or to detect poor designs.

This paper will describe how an automated system for structured analysis and design, **team*work***®, can be integrated with an APSE to support software systems development from specification through implementation. These tools complement each other to help developers improve quality and productivity, as well as to reduce development and maintenance costs. Complete system documentation and reusable code also result from the use of these tools. Integrating an APSE with automated tools for structured analysis and design provide capabilities and advantages beyond those realized with any of these systems used by themselves.

Introduction

Developing quality software on time and within budget has proven to be a difficult task. Statistics gathered by the government and private industry have shown that software development projects are difficult to control [Boehm 81]. This results in software systems that can be extremely expensive with less than adequate performance.

These problems have fostered several solutions. The U.S. Department of Defense performed an analysis of its software applications, concentrating on problems inherent with coding and implementation. This analysis resulted in the development of Ada [DoD 81]. Other people were addressing problems associated with software requirements. The results of this effort has resulted in the development of several software development methodologies based on the concept of a software lifecycle [DeMarco 78, Page-Jones 80, for example].

The DOD identified a problem specific to the *implementation* of embedded systems. There were a number of languages in use and there was potential that this number would continue to grow. The lack of a standard implementation language resulted in money being spent on new compilers (which were not significantly better), training and maintenance. The development of the Ada programming language was seen as an answer to this problem. In addition, the solution would include a programmer's environment, or toolkit, called the "APSE."

APSE

The Ada Programming Support Environment (APSE) was proposed to augment the Ada language [DoD 80, Stennig 81]. It includes tools such as the compiler, language sensitive editor, and debugger. These tools are designed with knowledge about the structure of Ada and are focused on the implementation phase of software development. The APSE presents a uniform development environment to aid Ada programmers.

APSEs help solve the problems of implementing embedded systems that were recognized by the DOD. A reduction in software development costs can be realized as a result of making the implementation phase more efficient. However, the problem still remains that APSEs do not thoroughly address the other phases of software development.

Software Development Lifecycle

Recent work has focused on gathering statistics from case studies of projects [Ramamoorthy 84]. At least half of the projects had problems which originated in the requirements or functional specification (see Figure 1). To help put this in perspective, we can view the software development process as divided into five (sometimes overlapping) phases: analysis, design, implementation, test and verification, and maintenance.

The analysis phase is concerned with understanding *what* a system is supposed to do. The result is supposed to be an *implementation independent* description or abstract view of the system to be developed. The product of analysis is a requirements specification (sometimes called a functional specification) that describes the system function and important constraints.

The design phase addresses *how* the system is to be implemented. It is concerned with the physical aspects of the system. The optimal structure of the various software modules and how they interface is determined. Ideally, the design information should be complete enough to reduce the implementation effort to little more than a translation to a target programming language.

The implementation phase is concerned with producing executable code. Knowledge of both the design and the target environment is incorporated to produce the final system software. All the physical aspects of the system are addressed during implementation.

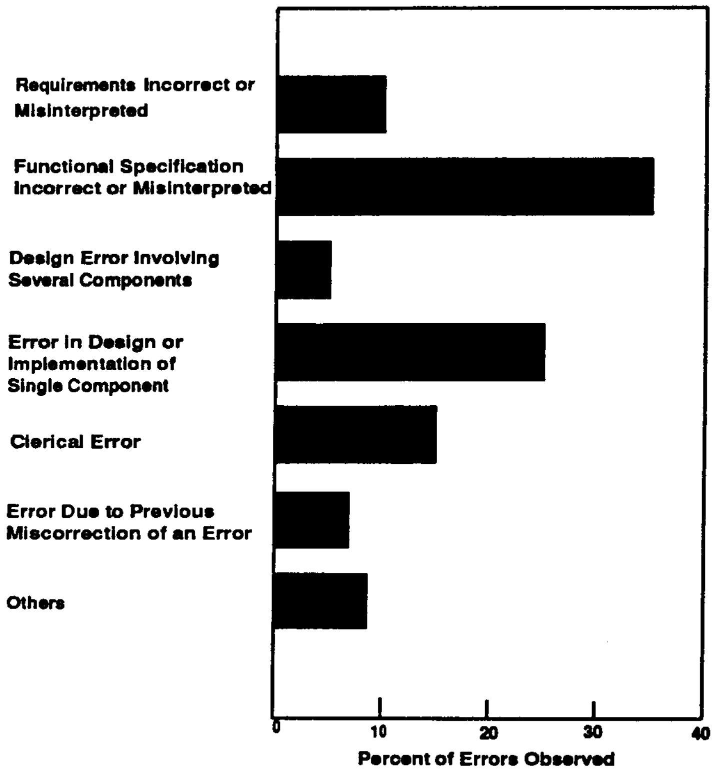


Figure 1: Sources of Errors[1].

Information from the previous three phases is used in the testing and verification phase. Test plans can be derived from specifications and designs [Boehm 84]. The testing phase verifies that the software conforms to the specification and that the code is correct. The best that test and verification techniques can do is prove that a program is consistent with its specification. They cannot prove that a program meets the user's desires [Wulf 80]. This means that extra care must be taken during analysis to insure that the specification is a complete and correct reflection of what the user really wants. This can be accomplished through methods that support checks for consistency and clearly communicate system requirements. Team*work*/SA supports one such method, and it will be discussed later in this paper.

Bug fixes and adaptations which result from experience with the software are activities of the maintenance phase. At this point the software is being used -- the ultimate test. Users will come across errors or suggestions as they gain experience with the software. Maintenance procedures must handle the orderly evolution of the code. They must insure that changes will not have deleterious effects on the system.

A study by [Boehm 84] showed that errors detected later in the development life cycle cost more to fix than errors detected during analysis (See Figure 2). Figure 1, discussed previously, showed that the majority of errors in a software project can be traced to requirements and specification problems. These facts illustrate the value of spending more time at the beginning of a project, performing analysis. This can be difficult for programmers and users to accept as both may be anxious to see code being produced [Ramamoorthy 84]. These ideas have only recently become well understood and brought into practice.

[1]Adapted from [Ramamoorthy 84]

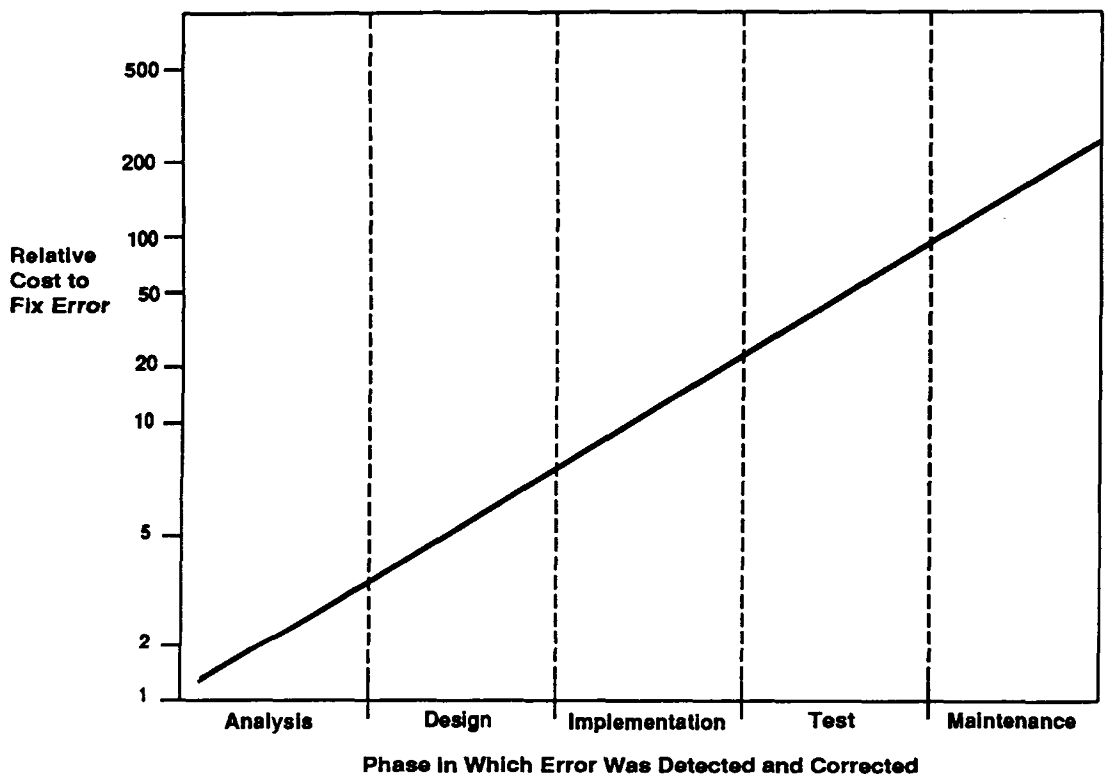


Figure 2: Cost of Error Versus When it is Detected[2].

Many approaches and methodologies utilize the concept of the software life cycle. In particular, structured analysis (which refers to several methods [Gane 79, DeMarco 78, Ross 77]) addresses the beginning phase of requirements analysis.

Structured Analysis

Structured analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements.

De Marco's approach [DeMarco 78] consists of the following objects: *data flow diagrams*, *process specifications*, and a *data dictionary* (See Figure 3).

Data flow diagrams (DFDs) are directed graphs. The arcs represent data, and the nodes (circles or bubbles) represent processes that transform the data. A process can be further decomposed to a more detailed DFD which shows the subprocesses and data flows within it. The subprocesses can in turn be decomposed further with another

[2]Adapted from [Boehm 84]

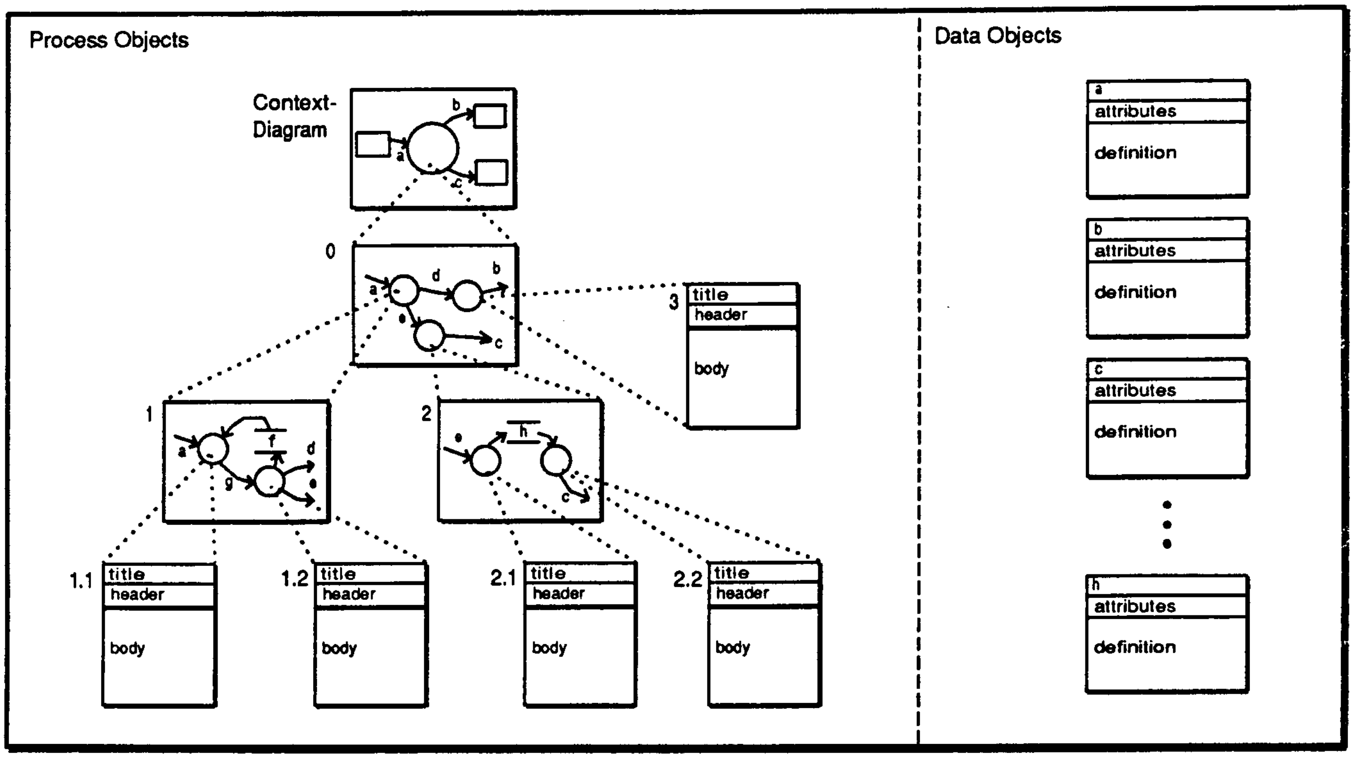


Figure 3: Analysis Model Objects

set of DFDs until their functions can be easily understood. Functional primitives are processes which do not need to be decomposed further. Functional primitives are described by a process specification (or mini-spec). The process specification can consist of pseudo-code, flowcharts, or structured English. The DFDs model the structure of the system as a network of interconnected processes composed of functional primitives.

The data dictionary is a set of entries (definitions) of data flows, data elements, files, and data bases. The data dictionary entries are partitioned in a top-down manner. They can be referenced in other data dictionary entries and in data flow diagrams.

Military standard 2167 [MilStd2167 85] requires that systems be specified in a top down manner using a structured approach similar to that described above. The high level of process and data abstraction inherent in structured analysis is compatible with the objectives of the Ada language. Where it is desirable to take an object-oriented approach to design [Booch 86, Cox 84], structured analysis helps to define classes and data hierarchies or data structure. For procedural approaches, structured analysis works well with structured design.

Structured Design

Structured design addresses the synthesis of a module hierarchy [Page-Jones 80]. The principles of cohesion and coupling are applied to derive a optimal module structure and interfaces. Cohesion is concerned with the grouping of functionally related processes into a particular module. Coupling addresses the flow of information, or parameters, passed between modules. Optimal coupling reduces the interfaces of modules, and the resulting complexity of the software.

Page-Jones' approach [Page-Jones 80] consists of the following objects: *structure charts, module specifications* and a *data dictionary.*

The structure chart shows the module hierarchy or calling sequence relationship of modules. There is a module specification for each module shown on the structure chart. The module specifications can be composed of pseudo-code or a program design language. The data dictionary is like that of structured analysis.

At this stage in the software development lifecycle, after analysis and design have been performed, it is possible to automatically generate data type declarations [Belkhouche 86], and procedure or subroutine templates.

Automating Structured Analysis and Design

Hardware CAD/CAM systems have contributed to the development a systems with higher levels of complexity, performance and reliability, at costs previously unattainable through purely manual design efforts. This is sparking interest in automating the software development process.

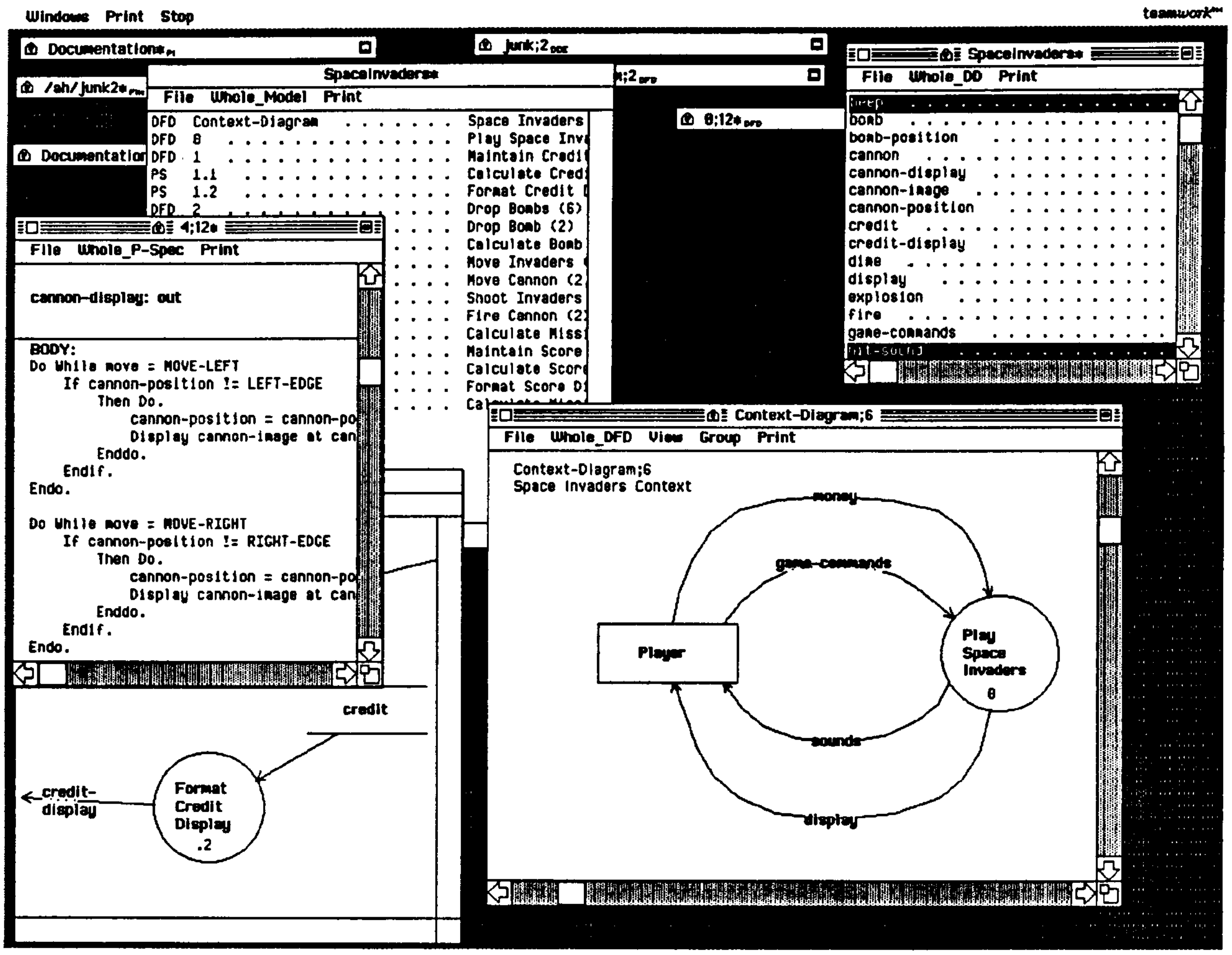
Team*work* is a set of automated tools for systems analysis and design. They can support many simultaneous users working on the same project or even many projects. They take advantage of features provided by the latest workstation technology, offering complete support of the DeMarco structured analysis techniques and the Page-Jones structured design techniques. Graphical diagrams are created using syntax-directed editors that incorporate model building rules. Its interactive graphics package supports a high resolution bit-mapped display, mouse and keyboard. Modern user interface techniques are used, including a multi-window display and context specific pop-up and pull-down menus.

Multiple, simultaneous views of a specification or a design can be displayed by **team*work*/SA** (See Figure 4). It has simple commands for traversing through the various parts of a model. Model objects may be entered in any order. The graphics editors allow diagrams to be easily produced and edited. Diagrams as well as components of diagrams are automatically numbered and indexed. These features eliminate many manual, time consuming tasks.

Project information is retained in a project library, through which individuals can simultaneously share model information and computer resources. Team members linked over the network can access the same information for review. Multiple versions of model objects are retained in the library. Team members can independently renumber and repartition diagrams, which allows exploration of different approaches to describe a system.

Team*work*'s consistency checker detects specification errors within and between data flow diagrams, data dictionary entries, and process specifications, and design errors within and between structure charts and module specifications. Typical errors and inconsistencies include DFD balancing errors (data flows from one diagram that do not match data flows to a related diagram) and undefined data dictionary entries. The consistency checker uses the semantics and rules of structured analysis and structured design. Checking is performed "on-demand", which allows the analyst and designer to work top-down, bottom-up, or any other way. It encourages the exploration of partial models that may be (during the intermediate stages of building the model) incomplete or incorrect. The speed and depth of checking in **team*work*/SA** helps produce consistent and correct specifications, which can be used with the tools provided in an APSE.

ORIGINAL PAGE IS
OF POOR QUALITY



```
DFD  Context-Diagram  . . . . . . . .  Space Invaders
DFD  0  . . . . . . . . . . . . . . .  Play Space Inv
DFD  1  . . . . . . . . . . . . . . .  Maintain Credi
PS   1.1  . . . . . . . . . . . . . .  Calculate Cred
PS   1.2  . . . . . . . . . . . . . .  Format Credit
DFD  2  . . . . . . . . . . . . . . .  Drop Bombs (6)
                               . . . .  Drop Bomb (2)
                               . . . .  Calculate Bomb
                               . . . .  Move Invaders
                               . . . .  Move Cannon (2
                               . . . .  Shoot Invaders
                               . . . .  Fire Cannon (2
                               . . . .  Calculate Miss
                               . . . .  Maintain Score
                               . . . .  Calculate Scor
                               . . . .  Format Score D
```

```
cannon-display: out

BODY:
Do While move = MOVE-LEFT
    If cannon-position != LEFT-EDGE
        Then Do.
            cannon-position = cannon-po
            Display cannon-image at can
        Enddo.
    Endif.
Endo.

Do While move = MOVE-RIGHT
    If cannon-position != RIGHT-EDGE
        Then Do.
            cannon-position = cannon-po
            Display cannon-image at can
        Enddo.
    Endif.
Endo.
```

```
beep  . . . . . . . . . . . . . . .
bomb  . . . . . . . . . . . . . . .
bomb-position  . . . . . . . . . .
cannon  . . . . . . . . . . . . . .
cannon-display  . . . . . . . . . .
cannon-image  . . . . . . . . . . .
cannon-position  . . . . . . . . .
credit  . . . . . . . . . . . . . .
credit-display  . . . . . . . . . .
dime  . . . . . . . . . . . . . . .
display  . . . . . . . . . . . . .
explosion  . . . . . . . . . . . .
fire  . . . . . . . . . . . . . . .
game-commands  . . . . . . . . . .
hit-sound  . . . . . . . . . . . .
```

```
Context-Diagram;6
Space Invaders Context
```

Figure 4: team*work*/SA Desktop

Integration of Teamwork with APSE

Team*work* was designed to allow the information it captures to be utilized for many purposes. These include packaged specifications, project status reports, configuration management, system documentation, and test plans. The information is captured as the specification and design are created. As described above, team*work* helps to insure consistency of the information as a system progresses through these phases. The relationships between the various representations of processes, data, and modules are recorded in the project library. This information may be selectively retrieved and reformatted with post processors which can be developed for a variety of software development tasks, such as the following:

- Producing data type declarations and procedure templates specific to the syntax of any language, especially Ada.
- Generating test plans.
- Generate formatted requirement documents, such as MIL-STD 2167.

In addition, by combining an APSE with team*work*, the complete lifecycle documentation can be consistently maintained, from requirements to code listings. If any change is made to any piece of a project, that change can be reflected in the corresponding parts of the project.

Conclusion

APSEs help reduce some of the problems associated with software development, especially during the implementation phase. Automated analysis and design environments address the problems associated with poor specifications and software system structure. Either tool by itself is better than totally manual development. The combination of all these tools can provide automated support for the entire software development lifecycle, insuring consistency and reducing errors and developments costs.

References

[Belkhouche 86] Belkhouche, B., and J.E. Urban.
Direct Implementation of Abstract Data Types from Abstract Specifications.
IEEE Transactions on Software Engineering :649-661, May, 1986.

[Boehm 81] Boehm, Barry W.
Software Engineering Economics.
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Boehm 84] Boehm, Barry W.
Verifying and Validating Software Requirements and Design Specifications.
Software , January, 1984.

[Booch 86] Booch, G.
Object-Oriented Development.
IEEE Transactions on Software Engineering :211-221, February, 1986.

[Cox 84] Cox, Brad J. .
Message/Object Programming: An Evolutionary Change in Programming Technology.
Software :50-61, January, 1984.

[DeMarco 78] DeMarco, Tom.
Structured Analysis and System Specification.
Yourdon Press, New York, 1978.

[DoD 80] US Dept. of Defense.
Requirements for Ada Programming Support Environments - Stoneman.
February, 1980

[DoD 81] US Dept. of Defense.
Reference Manual for the Ada Programming Language - Proposed Standard Document.
July, 1981

[Gane 79] Gane, Chris and Trish Sarson.
Structured Systems Analysis: Tools and Techniques.
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1979.

[MilStd2167 85]
Military Standard - Defense System Software Development DOD-STD-2167.
June, 1985

[Page-Jones 80] Page-Jones, M.
The Practical Guide to Structured Systems Design.
Yourdon Press, New York, 1980.

[Ramamoorthy 84]
Ramamoorthy, C.V., et. al.
Software Engineering: Problems and Perspectives.
Computer :191-209, October, 1984.

[Ross 77] Ross, D. and R.E. Schoman Jr.
Structured Analysis for Requirements Definition.
IEEE Transactions on Software Engineering SE-3(1), January, 1977.

[Stennig 81] V. Stennig et. al.
The Ada Environment: A Perspective.
Computer :26-36, June, 1981.

[Wulf 80] Wulf, W.A.
Trends in the Design and Implementation of Programming Languages.
Computer :14-23, June, 1980.

N89-16328

A Software Development Environment Utilizing PAMELA

Prepared by:

R. L. Flick
&
R. W. Connelly

Westinghouse Defense & Electronics Center
P.O. Box 746
Baltimore, Maryland 21203

May 12, 1986

A Software Development Environment Utilizing PAMELA[1]

Abstract

Hardware capability and efficiency has increased dramatically since the invention of the computer. while software programmer productivity and efficiency has remained at a relatively low level. A user-friendly. adaptable. integrated software development environment is needed to alleviate this problem. The environment should be designed around the Ada[2] language and a design methodology which takes advantage of the features of the Ada language such as the Process Abstraction Method for Embedded Large Applications (PAMELA).

Introduction

Since the invention of the computer. advances in software development productivity have not kept pace with hardware productivity. Although the throughput of modern computers has made a 1.000.000-fold increase over the last thirty years. software productivity has increased only slightly. During the same period. hardware costs have decreased dramatically and software costs have skyrocketed. Moreover. the complexity of embedded systems is growing exponentially. putting an ever increasing demand on software production.

Many studies have shown that the major costs in the software development life cycle occur after system delivery. Approximately 70% of these costs are incurred during the maintenance phase. There are several reasons for this:

1. Personnel costs for software professionals have risen steadily over the years. Consequently. for large systems designed to last many years. the cost of people becomes a major concern.

2. Inadequacy of documentation either internal or external to the code is a continual source of increased costs. Frequently on large systems. a modification in one routine will affect many other routines in unexpected ways. It is not uncommon that a

1. PAMELA is a trademark of Dr. George Cherry, Reston, Virginia
2. Ada is a registered trademark of the United States Department of Defense, Ada Joint Program Office.

change to correct one error will lead to numerous other errors. This is partially because large programs are intrinsically hard to understand, but also because inadequate documentation hampers understanding. Furthermore, most programming languages do not promote greater understanding since they do not always enforce good software engineering practices.

3. Yet another factor is an inadequate design process. Frequently, paper designs are created by systems engineers and then handed to programmers for implementation. The programmer often will tend to stray from the paper design in order to increase efficiency or make changes that are required by the constraints of the language employed.

The real cost of software therefore is in the maintenance of programs - but it originates in the methods and languages used to create these programs.

Current projections show that the cost of developing software is likely to continue to increase unless new, more efficient methods are employed. If current trends continue there will be a short fall of programmers by 1990 which may exceed 800,000.[3] Such a devastating short fall will slow software development to a crawl for many major government programs.

Current trends can be reversed by developing and utilizing standard software engineering practices throughout the software industry. These practices can be implemented in an expert system that is designed to specifically support one design methodology. In addition the methodology used must be specific to the language that is supported. The preferred language to be used is Ada, and one methodology that is specifically designed for the Ada language is PAMELA.

The Ada Language

Since the software development environment supports development of large embedded applications for the Department Of Defense (DOD) applications, and incorporate state-of-the-art tools, the language of choice is Ada. Ada is a fairly new language developed by the DOD specifically for embedded applications. Although Ada is new, the DOD has set a requirement that all new software written for the DOD will be done in Ada. As the advantages of using Ada as a general purpose programming

3. Mr. Edward Berard, EVB Systems, ACM SIGAda meeting, Los Angeles, California, February, 1986.

language become more fully developed, commercial firms will also choose Ada for there software needs. Some of the important features brought to software engineering are :

Code Reusability:

Ada supports code reusability in the form of generic packages, a common library of compiled units, and modular coding techniques. With these facilities Westinghouse has established a common database of program modules at the company level. By establishing and using this database of reusable software modules, generating software for embedded applications has become cheaper and faster.

Tasking:

Something new that is supported by Ada and virtually no other language, is the task unit. This unit is on the same level as a function or subprogram with one important difference: a task unit can be declared as a type. Because of tasking, generating embedded systems that require some form of parallel processing is easier.

Parallel Development:

An important feature of Ada, is the ability to do parallel development. Ada offers this facility in the form of separate compilation units. Westinghouse has found that several individuals can work on different sections of the code and not interfere with each other, and that code development is not dependant on any special order of accomplishment (other than Ada's dependency rules of course). Westinghouse has been able to increase software engineering productivity by reducing the scope of dependencies within the software application.

Information Hiding:

Ada provides the facility to hide the underlying machine dependent representation of data items. This discourages the software engineer from depending on a machine specific characteristic when implementing a section of the software system. It also means that the code generated should be transportable to any other machine that supports a validated Ada compiler.

Strong Type Checking Across Separate Compilation Units:

Ada is a strongly typed language that will not allow nonconformant data types to be passed between program units. The purpose of Ada's strong type checking is to prevent common errors from occurring when calling another software engineer's code.

Ada's Place In The Design Of A Software System:

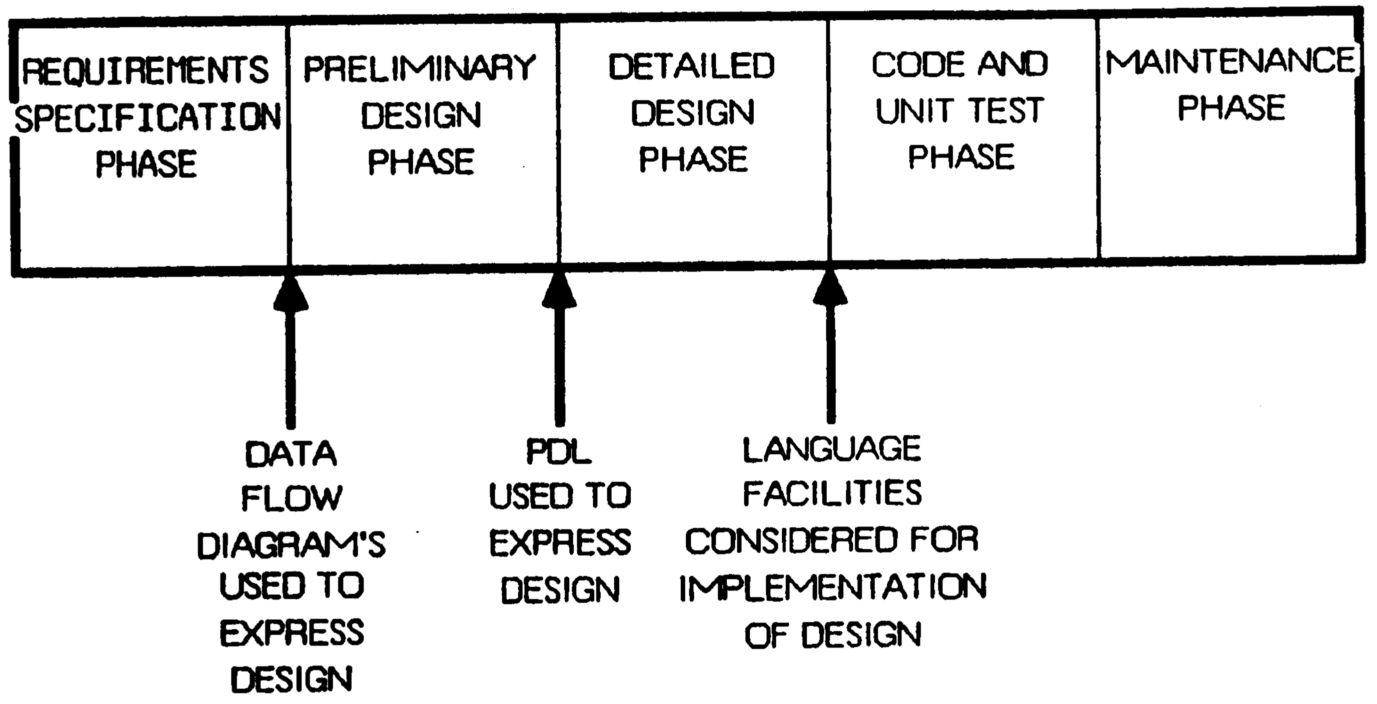
To be able to take advantage of the state-of-the-art facilities that Ada offers, the perception of when the capabilities of the programming language that is used is to be considered must be changed. In addition a design methodology that implements design concepts specified in MIL-STD 2167 and takes into account the improved facilities of Ada must be utilized.

Up until now the typical method for designing a software system involved specifying requirements, doing a preliminary design, doing a detailed design using a PDL and finally selecting a language and implementing the design. The primary methodology used when designing the system was typically a derivative of Data Flow Diagrams. (see Fig. # 1) This approach has worked with other languages (before Ada) because they did not provide sophisticated facilities for embedded environments such as tasking. Languages therefore, had little impact on the design of the system itself. All popular languages, aside from Ada, are sequential in nature. The design methodology used to express a system under development in this language is compatible with the capabilities of the language used and is sequential in nature.

If Ada is to become an effective alternative, several common practices and assumptions used in designing an embedded system must change, and a design methodology that is designed to accommodate a specific language must be used. To be able to take advantage of the advanced features that Ada offers, the methodology must take the language features into consideration in the preliminary design phase of a software system (see Fig. # 2). This means that consideration of language facilities should be an integral part of the preliminary and detailed design of the system. If the language considerations are made early in the development of the preliminary design, the overall impact will be in the areas of coding and integration time. These two areas comprise most of a software systems development cost. If however, Ada's facilities are not considered early in the preliminary development, Ada will offer almost no advantage over any other language.

The method of considering Ada's facilities in the preliminary and detailed design phase is dependent on the methodology used to express these designs. The popular methodologies of flow chart's, data flow diagrams, etc. will be of little advantage in the preliminary design phase when using Ada. The inability of these methodologies to express the unique facilities of tasking, code reusability, modular design, and parallel development diminish their usefulness for creating a design based on

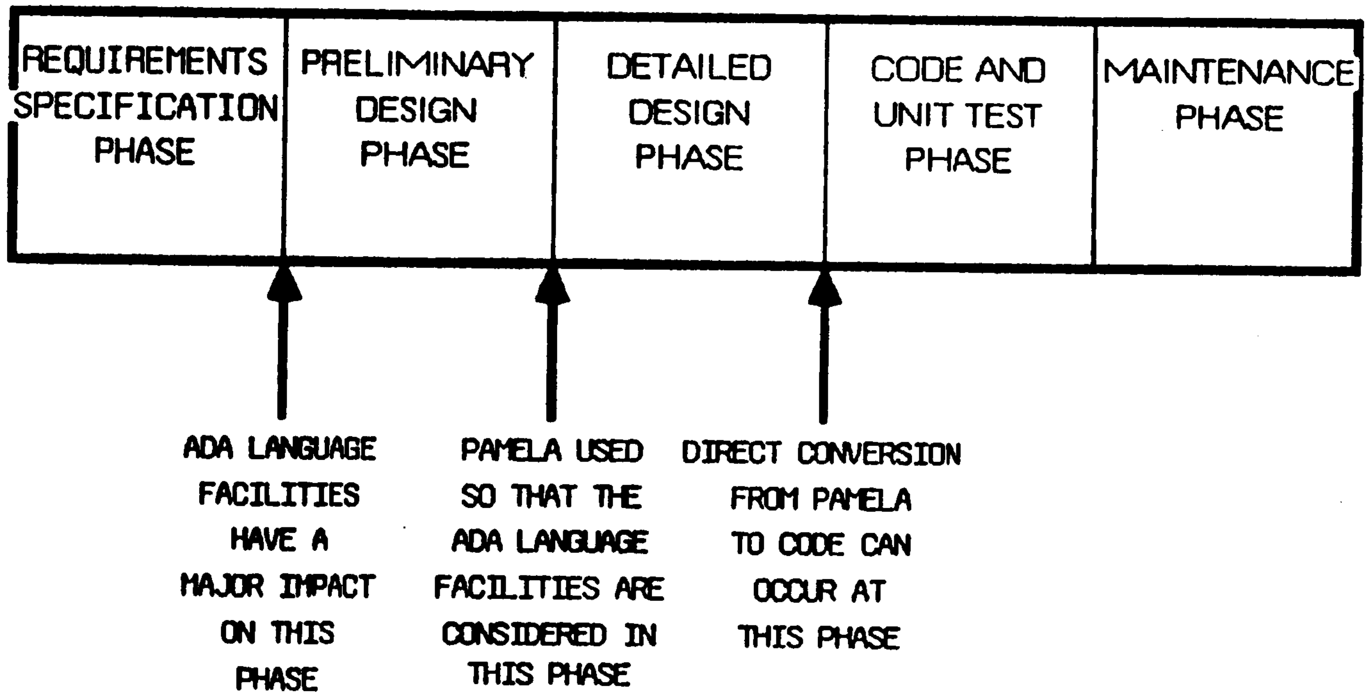
THIS METHODOLOGY REFLECTS THE SEQUENTIAL NATURE OF AVAILABLE LANGUAGES


REQUIREMENTS SPECIFICATION PHASE
PRELIMINARY DESIGN PHASE
DETAILED DESIGN PHASE
CODE AND UNIT TEST PHASE
MAINTENANCE PHASE
DATA FLOW DIAGRAM'S USED TO EXPRESS DESIGN
PDL USED TO EXPRESS DESIGN
LANGUAGE FACILITIES CONSIDERED FOR IMPLEMENTATION OF DESIGN

CURRENT METHOD OF SOFTWARE SYSTEM IMPLEMENTATION

FIGURE #1

THIS METHODOLOGY REFLECTS THE PARALLEL NATURE OF THE ADA LANGUAGE


REQUIREMENTS SPECIFICATION PHASE
PRELIMINARY DESIGN PHASE
DETAILED DESIGN PHASE
CODE AND UNIT TEST PHASE
MAINTENANCE PHASE
ADA LANGUAGE FACILITIES HAVE A MAJOR IMPACT ON THIS PHASE
PAMELA USED SO THAT THE ADA LANGUAGE FACILITIES ARE CONSIDERED IN THIS PHASE
DIRECT CONVERSION FROM PAMELA TO CODE CAN OCCUR AT THIS PHASE

SUGGESTED METHOD OF SOFTWARE SYSTEM IMPLEMENTATION

FIGURE #2

Ada. It is therefore necessary that a design methodology that can express parallelism, code reusability, modular design, and parallel development be used. It is also necessary that the methodology used in the preliminary and detailed design be a direct expression of the Ada language. We have found that PAMELA fits this description.

Description of PAMELA

PAMELA is a methodology for producing real-time Ada programs which utilize Ada tasking. It was designed by Dr. George Cherry (Reston, Virginia) to address the needs of Ada users in developing real-time programs using Ada's rich variety of language features.

PAMELA is a structured methodology that encourages a top-down approach, with each step in the method revealing more details than the previous step. (see Fig. # 3) It is also a graphical methodology which produces pictographs of the underlying Ada code. In fact, Ada package and task specifications, as well as skeletal package and task bodies, can be produced directly from PAMELA graphs.

PAMELA combines aspects the two most prominent program representation methodologies of the past two decades, data flow diagrams and control flow diagrams (flow charts). It's pictographs are very similar to those produced by structured analysis and structured design techniques (data flow diagrams), but it also embodies a certain amount of control flow information - primarily because of the well defined Ada tasking mechanism.

PAMELA guides the program designer in the selection of multiple, concurrent threads of execution (called **processes** in PAMELA nomenclature). By analyzing the requirements of the problem, and by following the process idioms outlined in the method, (see Fig. # 4) the program designer identifies which elements of the program should become processes. He then determines what kind of data or control signals must be passed between processes. Next, he determines which process is the producer of the flow and which is the consumer. (see Fig. # 5) Finally, he determines which of the processes should be single-thread (a typical C, PASCAL, or FORTRAN - style program) and which should be multi-thread (more than one Ada task). Once the graph has been annotated with this information, Ada code can automatically be generated (in skeletal form) which implements the design.

PAMELA

- Define External Entities
- Decompose Master Procedure Into Processes
- Map All I/O From Higher Level Graph to Decomposed Graph
- Identify Internal Interfaces Between Processes
- Identify Calling Process

FIGURE #3

DEFINITION OF A PROCESS

- Hardware — dependent, application — independent function
- Device driver
- Interrupt handler
- Buffer between asynchronous activities
- Application — dependent, hardware — independent function
- Proprietor (monitor)
- Cyclic activity
- Activity requiring a distinct priority
- Vigilant activity
- Activity that must wait a specified time for an event

Reference: Process Abstraction Method for Embedded Large Applications
George W. Cherry, Reston VA 1985.

FIGURE #4

LEVEL 1

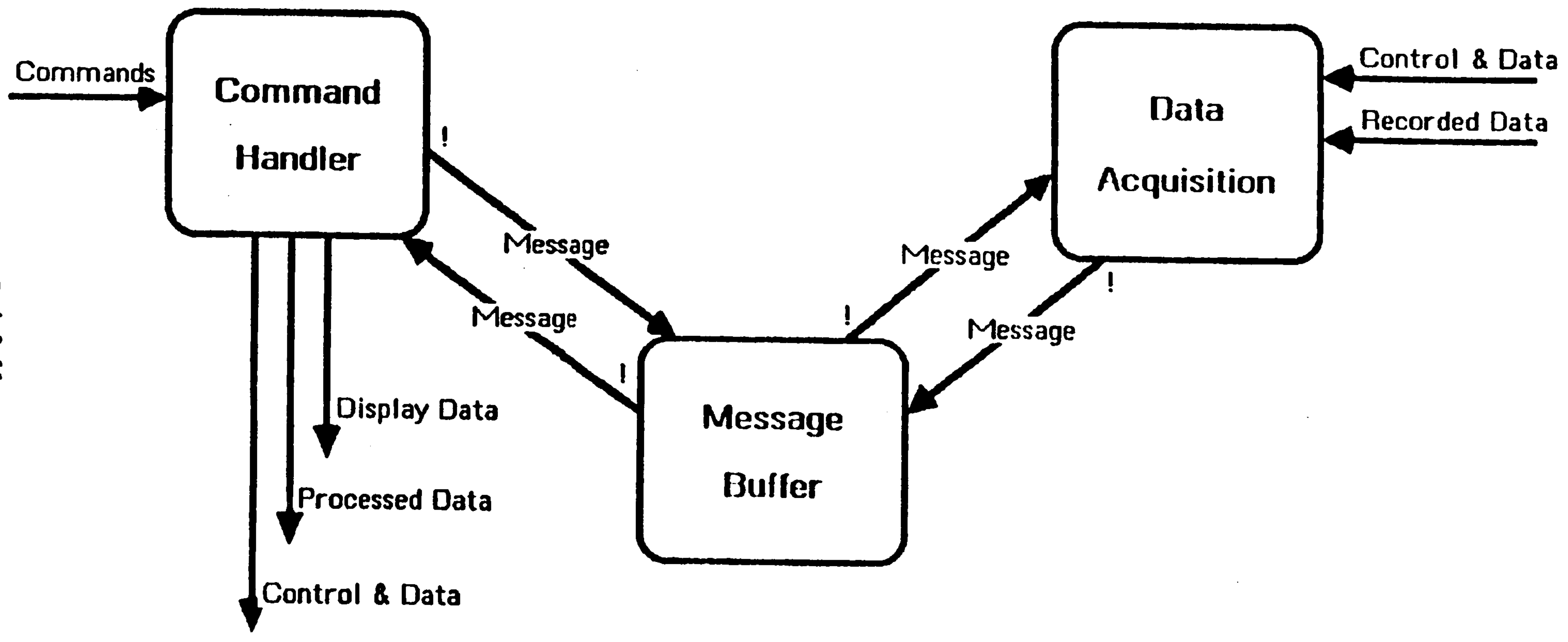


FIGURE #5

From the PAMELA graphs, single-thread processes become Ada tasks, while multi-thread processes become packages. The package body of a multi-thread process contains task and package specifications for the lower level single- and multi-thread processes respectively.

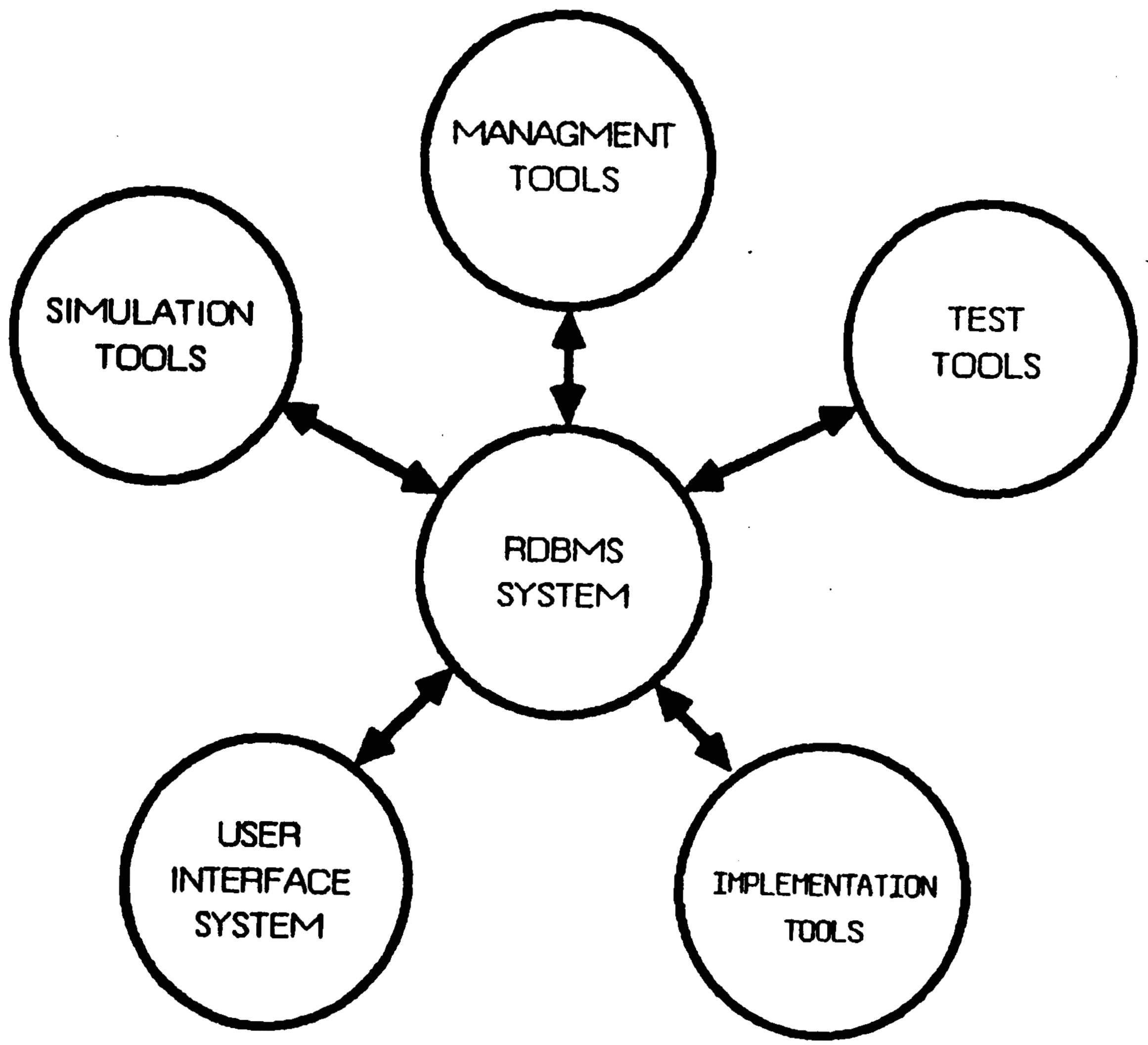
The Problem To Be Solved

The problem of increasingly large and complex software systems, in concert with a massive projected shortfall of software engineers in the next decade, fueled by spiraling software costs, must be abated. It is foolhardy to think that software systems will decrease in complexity; all current trends support the notion that future software systems will be very much more complex than those of today. The number of software engineers may increase by the next decade, but probably not fast enough to meet the challenges of these more complex systems. If software engineers continue to be in high demand, there is little hope of abating spiraling software costs.

The key to the solution of the problem is to substantially increase the productivity of software professionals. The primary tool to accomplish this goal is a high performance Software Development Environment (SDE). The SDE must be designed and built around a single specific language and design methodology. Since the DOD has mandated that all new software written for the DOD will be in Ada, Ada is the natural language of choice for the SDE. There are several new Ada based design methodologies such as Object Oriented Design (OOD), PAMELA (Process Abstraction Methodology for Embedded Large Applications), and Ada Partition Programming Language (APPL). Of all the new design methodologies we are considering PAMELA, as an example, around which to design the SDE.

The Software Development Environment

A software development environment (SDE) is being created at Westinghouse which supports all activities associated with the development of embedded software systems, as well as software management and post deployment support. By integrating all of the activities involved in software development under the control of one expandable, adaptable environment, software development and support can be made easier, more cost effective, and more reliable (see Fig. # 6).



SOFTWARE DEVELOPMENT ENVIRONMENT SYSTEM DIAGRAM

FIGURE #6

Important elements of the environment are:

Reliability:

The reliability of programs created under the SDE must be significantly greater than that of programs generated without such an environment. Reliability metrics, when applied to programs created under the SDE, should show a measurable and statistically significant increase in reliability. This in turn will require that the SDE itself be an exceptionally reliable program. We have seen that by using PAMELA, it is relatively straightforward to create reliable designs in a timely manner. Since the underlying Ada code maps directly to PAMELA pictographs, it is only necessary to correctly identify control and data flows at a high (pictograph) level to insure the reliability of the underlying code. Application of expert system techniques will also enhance the reliability of the environment.

Ease of Use:

The SDE will encompass a common, multi-level, user friendly interface. In particular, the interface will be as easy to use for the novice as for the expert. This will probably be accomplished with a multi-window, menu-driven interface which will provide full prompting for the novice. For the expert, a series of function keys and/or control keys can be defined (by the environment and/or by the user) to enable rapid execution of frequently used command sequences. For others, on-line help and an English-like command interface will be provided. Every user will be able to select the interface he/she prefers and will also be allowed to jump to any particular interface level at will.

PAMELA will support the ease of use concept since it is graphical, and is supported by an interactive, full screen tool which can automatically generate executable code.

Cost Effectiveness:

The environment should be networked so that individual workstations can be utilized by development and management personnel. This means that each individual or team will be able to achieve maximum utilization of the facilities available while avoiding the typical slow down experienced with multi-user super-mini implementations. Because of advances in micro-processors, a single user workstation can provide an engineer with a more responsive machine than can normally be attained with a time-shared super-mini. The resultant increase in throughput, can increase productivity substantially. As a side benefit, costs incurred due to main CPU down-time can be minimized by allowing the workstations to operate independent from the host.

The use of PAMELA should also prove cost effective in that it allows for rapid prototyping of the software system within the SDE. This allows the program implementors (and designers) to identify and correct potential or unexpected problem areas before they actually become problem areas.

Adaptability:

The environment will support various tools that will measure productivity, quality, maintainability and overall cost. This means that management will have the ability to measure all aspects of the evolving system in terms of quality, maintainability and cost. It will also allow the measurement of team performance compared to calibration data contained in the database. Such measurements can be used to recalibrate the system to more accurately reflect real world situations.

PAMELA has proven itself to be quite adaptable. In one particular instance, a 7000 line program was re-designed and re-implemented from scratch in just three days.

Design Continuity:

The environment is an expert system which provides tools that enhance all phases of the software life cycle. Program requirements are entered into a relational data base under the control of the expert system. Once a requirement has been entered, a basis is established for all later phases of the software life cycle. In particular, design, coding, and test specifications are derived from the requirements and related back to them by the expert system. This provides traceability from requirements to code, but also allows the environment to provide an impact analysis report for each requirement.

Software designs (specifically PAMELA designs) are accepted by the expert system. Once a design has been entered, it can be verified for compliance to the requirements by the environment.

Tools such as language sensitive editors, compilers, and debuggers which facilitate the coding and unit testing process can also be directed by the environment. For example, a compiler which produces diagnostic information could relate the number and kinds of programmer errors to the environment. The expert system could in turn relate this information back to a language sensitive editor to help correct programmer mistakes as they happen.

Since all requirements and design information are entered into the expert environment, test scenarios and/or test cases can be automatically generated to verify the design.

Program Visibility

The SDE supports all levels of management visibility into the current status, and projected results of the project. This means that the management functions of progress tracking, scheduling, and cost information gathering will be provided by the expert system. This includes but is not limited to, the automatic generation documentation and management reports with little or no human intervention. In addition there is some capability of the software factory concept in that generic, reusable software could be placed in the design by the expert system itself. This will alleviate the problem of the software engineer overlooking a reusable package that is in the database of reusable program modules.

Projected Environment Layout:

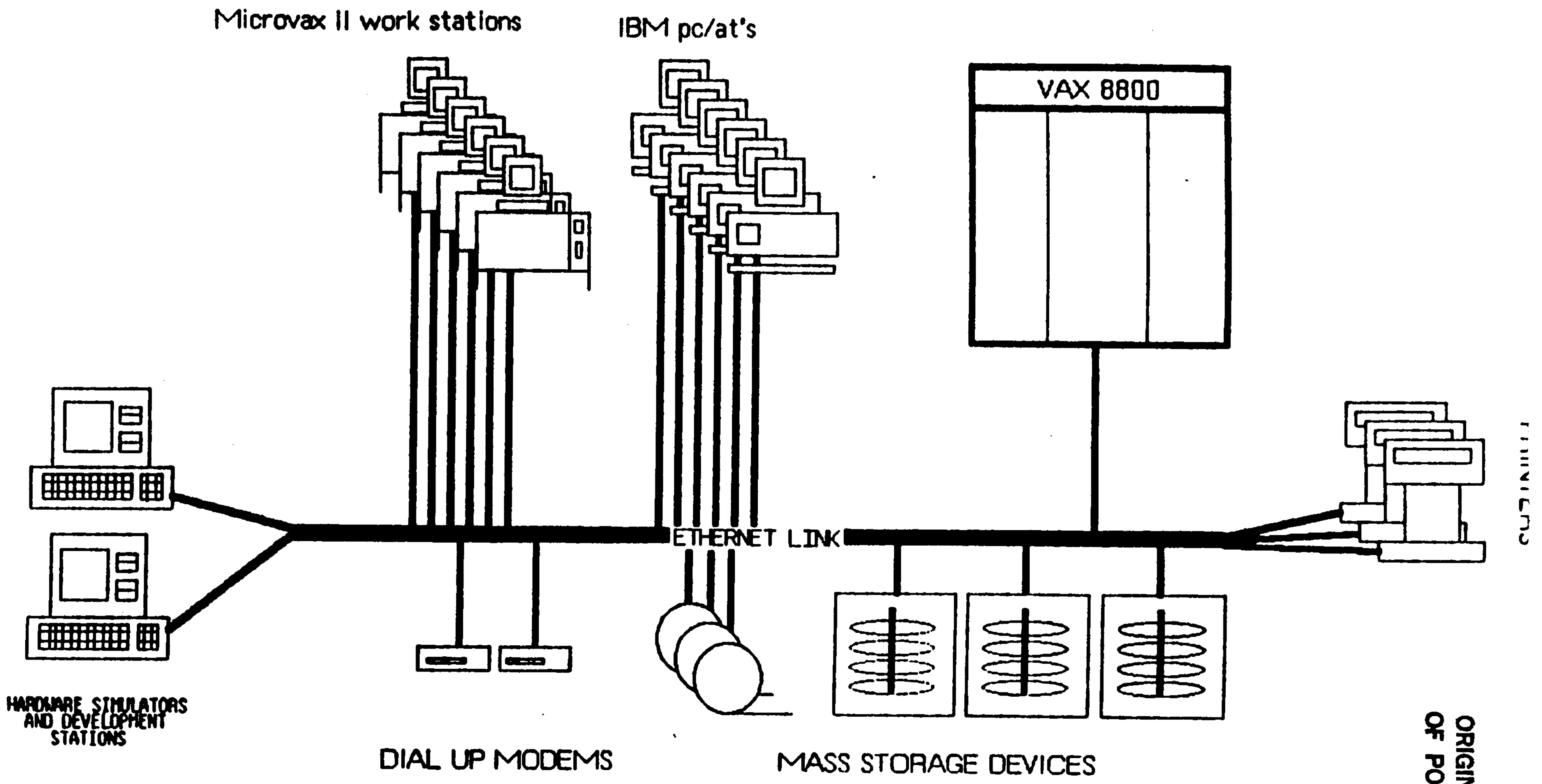
The environment will be as flexible as possible and support all types and sizes of software development. The system will incorporate artificial intelligence, networking, database management and some form of electronic mail. The hardware of such a system is projected to be composed of the following components:

1. A VAX minicomputer as the central database machine.
2. Several VAXSTATION II's as individual workstations.
3. Several micro computers such as IBM PC/AT's for manager workstations.
4. Some type of clustering system.
5. Some type of LAN (Local Area Network) system for node communications.
6. Hardware simulators and development stations for hardware specific support.

(see Fig. # 7)

Potential Problem Arenas

As is the case with all things, PAMELA is not perfect. There are two potential problem arenas associated with PAMELA which affect the performance of the SDE. For one thing, PAMELA designs typically


Microvax II work stations
IBM pc/at's
VAX 8800
ETHERNET LINK
HARDWARE SIMULATORS AND DEVELOPMENT STATIONS
DIAL UP MODEMS
MASS STORAGE DEVICES

ORIGINAL PAGE IS OF POOR QUALITY

THE SOFTWARE DEVELOPMENT ENVIROMENT HARDWARE ARRANGEMENT

FIGURE #7

create too many tasks. This is not a fault with the methodology per se. but reflects the fact that there are precious few machines out there that are made to run Ada. The methodology has been altered somewhat to account for this fact. but in so doing. it has lost some of its "virtual machine" flavor.

Another potential problem arena is that of testing. The current suggestion is to test each single-thread process using current structured techniques. As each is tested. it is integrated with the others and an integration test is performed. Eventually a multi-thread process will be declared valid and it then can be integrated with other processes. There is no method however for verifying that all the task rendezvous and other task interactions are correct. This is still a matter of art as much as it is of science but may be alleviated somewhat by the use of heuristic approaches common in expert systems. It is not clear however. whether this will be harmful for large embedded systems. If the paper design is solid. the implementation should be as well: but there is unfortunately no method for verifying paper designs either.

Potential Solutions

The horizon should not be clouded by the concerns raised above. Each problem poses new and exciting possibilities for new technologies and new ideas to solve those problems. Each new challenge brings us closer yet to another breakthrough.

The problems posed in the development of a state-of-the-art software development environment can be solved by hard work and dedication. They should not be attacked alone. but in concert with concerned organizations willing to lead us into the next century.

Conclusions

In conclusion. the need for a comprehensive. integrated software development environment has been demonstrated by the severe lack of productivity in developing software as compared to computer hardware. The need to automate documentation so that it provides a better picture of the program is essential to decreasing the maintenance costs of large software systems. An automated. integrated environment supporting a single specific language such as Ada and designed around a specific methodology such as PAMELA will reduce time and errors in the design and testing phases. Since the environment will ensure adequate tracking of requirements. design. implementation and testing. the cohesion is provided to aid management tracking of progress during the software life cycle. A common, multi-level, user friendly interface is absolutely required to insure maximum effectiveness for all users of various levels of experience and expertise. Finally, PAMELA is an ideal design methodology for such an environment, since it is Ada-based, and naturally addresses multiple concurrent tasks. PAMELA has been used on projects at Westinghouse and has proven its effectiveness for rapid prototyping, ease of design, maintainability and adaptability.[4]

4. Some material contained in this document was presented by Rich Connelly and Barbara Sullivan at the SigAda conference held in Boston Ma. in Nov. 1985

N89-16329

THE BENEFITS OF BOTTOM-UP DESIGN

Gregory McFarland

Grumman Data Systems
1000 Woodbury Rd.
Woodbury, N.Y. 11797

ABSTRACT

This paper examines an inconsistency in generic 'top-down' design methods and standards employed in the implementation of reliable software. Many design approaches adopt top-down ordering when defining the structure, interfaces, and processing of a system. However, strict adherence to a top-down sequencing does not permit accurate description of a system's error handling functions. The design of a system's response to errors is becoming critical as the reliability requirements of systems increase. This paper describes how top-down methods such as Object Oriented Design and Structured Design do not adequately address the issues of error handling, and suggests using a bottom-up substep within these methods to eliminate the problem.

1. INTRODUCTION

This paper describes the inability of top-down design techniques to allow for accurate design of the error handling features of a system. The primary concern involves what is tentatively termed the 'detailed design phase' of the software development process. This is the portion of the design process which provides a description of the system used as input to the implementation phase of the software life cycle. We believe that this design must accurately describe all the intended operations of the system to avoid the risk of 'interpretation' by programmers. Our discussion will make it clear that strict top down design techniques do not provide the designer an opportunity to specify the error handling features of a system. Acknowledging the mounting interest in 'structured design methods,' we must be certain that these methods address all of our requirements as designers, and that adopting them would not preclude certain design decisions. Additionally, the heightened reliability requirements of our systems necessitate that design methods provide the opportunity to address error handling issues.

Section 2 describes the software design environment we are considering. Section 3 defines techniques, standards, and tools often applied in the detailed design phase. The software design process is investigated by examining the activities performed during that effort in section 4. Section 5 details the problem encountered when using generic top-down methods in relation to the design of error handling facilities. Finally, section 6 describes how a bottom-up substep can be incorporated into existing methods to eliminate the problem.

2. SOFTWARE DESIGN ENVIRONMENT

For purposes of this paper, we will adhere to the definitions for 'life cycle' and 'method' found in [MCDE84]. The software life cycle defines a series of system views, each progressing from the abstract to the more concrete. A development method is concerned with the activities on one or more of these levels and comprises three distinct pieces: notation, guidelines, and analysis. The guidelines define rules for transforming the system at the previous level to the system at the current level. The current level is expressed in the notation defined by the method. Analysis is used to verify consistency within a level as well as that between levels.

A software development effort includes selection of a method to be applied in each life cycle phase. As indicated in the introduction, we are primarily concerned with the 'detailed design phase' where a representation of the system that can be used as a baseline for the coding or implementation phase is produced. According to the above definition of 'method,' few design techniques described in the literature today are 'methods.' Quite often only guidelines and/or notation are defined. Analysis techniques are rarely included. Additionally, individual efforts will normally modify the notation used based on past experience and tool availability. For this paper, we will concentrate our attention on the guidelines portion of the method. Therefore, we will assume that the final notation of the system after this phase is some form of Ada* PDL, that a PDL processor or Ada compiler is utilized to verify internal consistency, and some sort of structured design review is employed to verify the correctness of the resulting design in relation to previous design phases. We do not preclude the use of graphics during the design process, or as an additional output, but it will be the PDL that the programming staff utilizes during the implementation phase, and therefore this will be the final design notation. The final PDL representation of the system typically will define the system's modular structure, its data, and the processing to be performed by each module.

* Ada is a registered trademark of the U.S. Government, AJPO

3. SOFTWARE DESIGN TECHNIQUES, STANDARDS, AND TOOLS

The software development process is a complex combination of techniques, standards, and tools. Techniques are defined by the selected method and dictate the design steps. Standards are often dictated by contracts and impose additional constraints on the process. Tools can be automated aids such as editors, or logical tools such as the use of abstraction or information hiding. The combination of the various techniques, standards, and tools involved in each part of the design process can lead to problems like those described below.

Many design techniques found in the literature impose a top-down order of work within the level or phase where applied. The examples we will discuss are Object Oriented Design [OBJE85] [BOOC83] and Composite (Structured) Design [MYER78]. Both of these methods are 'top-down' since they require recursive application of the technique on the modules or operations that were defined in the previous step. In the case of Object Oriented Design, once the objects and operations have been defined, the designer must define the interfaces to these operations, perform a stepwise decomposition of the highest level module, and then repeat the entire design process for the newly defined operations. The stepwise decomposition of the highest module defines the interaction of this module with the newly defined operations. The implementation of these operations is not considered; they are 'abstractions.' Structured design incorporates a similar set of tasks for the design process, the main difference being the rules (guidelines) used to define the modules that 'implement' the current module. In structured design, only the structure of the system is defined. No method for defining the algorithmic portion of each module is proposed. If the technique employed to define each module's implementation section applies a top-down approach, then the entire detailed design phase is considered top-down.

Additionally, DoD standards and guidelines [DOD] for developing software systems impose a top-down structure on the development process. Unless alternate development techniques are approved by the contracting agency (see [SDST85]), top-down design, top-down coding, and top-down testing are required. As will be argued in the remainder of this paper, the use of a top-down ordering of the entire detailed design process is not desirable.

Many design techniques, including the two above, employ 'abstraction.' Abstraction is a valuable tool of the software engineer, but will be shown to be inappropriate if used throughout the entire detailed design phase. Abstraction allows designers to ignore the implementation details of 'other' parts of the system. This is useful during a decomposition process, but will lead to problems when connected with the design of a system's error handling facilities.

We will see how the combination of the above three items, top-down design techniques, contractual standards, and the utilization of abstraction, leads to problems when designing the error handling facilities of a system. A bottom-up approach may be applied during one substep of the overall detailed design process to eliminate this problem.

4. SOFTWARE DESIGN PROCESS

Consider the activities that occur during a typical detailed design effort. The selected method defines a set of guidelines which describe the steps a designer must undertake during the design process. As stated above, the design at this level typically includes module definitions, their relationships with each other, data definitions, and a description of the processing each module should undertake. The generic top-down design techniques being considered proceed as follows. First, select an undefined module and follow the guidelines specified by the technique. These guidelines result in additional modules and data definitions being defined. Second, determine the interfaces of these new modules and data objects. The guidelines may then suggest one of two possibilities. In the case of Object Oriented Design, stepwise refinement or some other technique is adopted to define the processing of the module. Once this is accomplished, the method is recursively applied to any resulting modules too large to be described as a single unit. An alternative approach, which might be found in a Structured Design, would be to first repeatedly apply the method to any undefined modules, completely defining the modular structure of the system and the interfaces to these modules. Once the entire system is decomposed, each module's processing is described, most likely in a top-down order.

Abstraction plays a large role in these top-down techniques. Abstraction permits the designer to utilize the interface information of other modules in the design of any module's implementation section. A hierarchy of modules is often viewed in a top-down fashion, with each module taking an abstract view of lower level modules in its 'implementation section.' The application of abstraction implies that only the interface information is needed for correct use of a module. Top-down implies that interface information for any module is used prior to that module having its implementation section defined. Thus we are relying on the premise that the design of any implementation section will not alter the interface of a module. In the case of error handling, this may not always be true.

5. THE PROBLEM

The problem associated with top-down design techniques and the use of abstraction becomes evident when considering the design of a module's processing section. This design will utilize prior design work that has identified interfaces and functionalities of subordinate modules. In other words, this processing section's design is based on the abstractions provided by the subordinate modules. Thus, the correctness of this design relies on the premise that these interfaces or abstractions will not change. While change is a natural part of the design process, attributable to designers' discovery of new information and backtracking to modify prior design decisions, change and backtracking should not be a direct consequence of the method used. Two assumptions concerning the error handling facilities of a module, which will be justified below, are that these facilities will not be known until the module's implementation is designed, and that these facilities will change the interface of the module. Based on these two assumptions, the design of every implementation section may change the associated interface. Therefore, the design of the processing section described above may become invalid when the subordinate modules' processing sections are defined. Since a top-down order of design is being employed, every processing section that causes changes in the associated interface, invalidates the assumptions used to design the processing section of superior modules.

First, the assumption that the error handling facilities of a module will change that module's interface should be considered. Errors can not be handled entirely within the module where they are generated. If errors were always handled locally, either no real error processing or correction would be performed, or each module would require knowledge of its actual use or purpose. Thus, either the systems will not be tolerant of errors, or the individual software within the system will not be general or reusable. For these reasons we will allow and even encourage that errors be propagated from modules and be handled where it is most appropriate. Now consider that a complete design, at the detailed level, will specify the potential error situations as well as the desired response to those errors. Errors may be propagated into or generated by a module. Depending on the error handling facilities provided in the chosen language, errors may or may not be gracefully handled. Consider the Ada programming language which provides extensive error handling facilities. In Ada, errors may be handled by special sections of code, and propagated out of the current module. The processing performed in response to errors changes the functionality or effect of this module. The possibility of errors being propagated out of a module also changes the interface of the module. Thus, the error handling facilities of a module add to or change the module's interface.

Consider also when the designer will be making decisions about the error handling of some module. Abstraction plays an important role in the application of the design method. Modules are defined in terms of their function and interface, while their implementation is not considered. These modules are then utilized during the design of the processing sections of superior modules. During the definition of a module's function and interface it is possible to define certain error situations that may arise. However, defining the internal response to these errors would imply that the designer is considering the implementation details of the modules. This is a violation of the abstraction principle and is inappropriate. Additionally, designers can not be cognizant of all the possible errors a module may generate. These errors will be discovered during the design of that module's implementation section. Accordingly, at the outset, the response to these errors will also be unknown. Therefore, there is a considerable potential that the interface of a module will be changed after that interface has been defined and used during earlier design activities.

The basic flaw described above is a consequence of the designer's reliance on the abstractions of other modules. The principle of abstraction has proven very useful in defining the structure of a system. However, it generally does not apply to the entire design process. It is unwise to design the implementation section of a module based on a number of abstractions if there is a likelihood that the abstractions will change. Doing so creates the potential for considerable rework and deviation from contractual standards and procedures.

The assumption made above that "a complete design, at the detailed level, will specify the potential error situations as well as the desired response to those errors," should be discussed. The content of a detailed design is a subjective decision. The life cycle phase considered in this paper, labeled 'detailed design,' was more accurately defined as the phase prior to implementation. Thus, the output of this phase, a description of the system in the selected notation, will be given to a programming staff for purposes of implementation. Alternatives to the above assumption are to not specify the error handling facilities to be incorporated by the system, or to specify them only partly. Consequently, the programmer must decide between not including any error handling facilities since they were not defined, or in the case of Ada, providing a general error handler that catches any error raised in or propagated to a module. Neither of these situations is desirable if reliability is a goal of the software. Alternatively, the programmer may handle those errors which he determines are generated by this module on an individual basis, deciding what processing is appropriate for each, and which should be propagated to calling modules. This will cause a module's implementation to deviate from its assigned function and interface. Finally, the programmer may perform the necessary work to make the following determinations:

1. Which errors may be propagated into the module?
2. What processing has already been performed in response to these errors?
3. What errors may be generated by this module?
4. What processing is necessary in response to both types of errors? and
5. Which errors get propagated out of this module.

This alternative requires communication between programmers and additions to the functionality and interface of the modules. None of these alternatives is as attractive as having the error handling facilities defined during the design process.

6. A SOLUTION

A simple solution to this problem is to design the processing sections of a system's modules in a bottom-up order. As each module has its processing section designed, appropriate changes can be made to the interface and functional description of the module. Thus, higher level modules utilize a more complete description of lower level modules. Performing this bottom-up substep within a design phase is compatible with both Object Oriented Design and Structured Design. This substep only requires that implementation sections are not designed until the structure and data definitions of the entire system have been defined. Once this is accomplished, the bottom-up order of processing section design may begin.

A bottom-up design order does not define any additional guidelines for the design of the error handling facilities of a system. At most, this will allow the designer the opportunity to consider the issue, and specify the required functionality prior to when that information is used in other design work. This will reduce the amount of change and wasted effort that results from basing design decisions on incomplete information.

7. SUMMARY

This paper defines a problem engendered by the top-down structure imposed by software design methods and standards applied during the detailed design of a software system. Designers whose techniques rely on abstract modules defined in a top-down order will find that the design of the implementation section of these modules will result in changes to their interfaces attributable to error situations defined, handled, and propagated. Changes to these interfaces invalidate assumptions made by higher level modules' implementation sections. One solution is to design modules' implementation sections in a bottom-up order, making the necessary changes to the interfaces of the modules.

This paper is not meant to criticize current methods or imply that they should be abandoned. Instead, it criticizes the ways in which these methods are applied. What is desired is an understanding that application of 'design methods' does not solve all the problems of software design. In addition to being executed correctly, design methods must be applied only where appropriate. Careful analysis is needed to determine what must be accomplished during each phase of the software life cycle, and how well the selected method or methods address these needs. It will often be found that existing design methods can not address all the activities required within even a single phase of the life cycle. For this reason, methods must be augmented with additional techniques or considerations to ensure the design process is complete and correct. The example described in this paper demonstrated that the design of error handling facilities of a system is not adequately addressed by generic top-down methods. Thus, special consideration is required to ensure that the overall design approach addresses this portion of the software system.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the encouragement and helpful comments of John D. Litke.

REFERENCES

[BOOC83] Booch, Grady; Software Engineering With Ada; Benjamin/Cummings Publishing Company, Inc., California, 1983.

[DOD] DOD-HBK-287 Defense Department Software Development Handbook

[DOD] DOD-STD-2167 Defense System Software Development

[MCDE84] McDermid, John, and Ripken, Knut; Life Cycle Support in the Ada Environment; The Ada Companion Series; Cambridge University Press, Great Britain; 1984.

[MYER78] Myers, Glenford, J.; Composite/Structured Design; Van Nostrand Reinhold Company, New York; 1978.

[OBJE85] Object Oriented Design Handbook; EVB Software Engineering Inc., 1985.

[SDST85] SDS - Tailoring Provisions for Ada; Technical Report E-98044; Electronic System Division, Air Force Systems Command, Hanscom Air Force Base, Bedford, MA, April 1985.

The Ada Object-Oriented Approach

Steve Nies
Harris Government Systems Sector, Software Operation
5Ø5 John Rodes Blvd., Bldg. 1
Melbourne, FL 329Ø2
(3Ø5) 242-5383

As systems become more complex, the cost of developing software is rising dramatically. Although the requirements phase has been identified as crucial for the successful development of large, reliable, systems, much of the software engineering effort to date has been in improving programmer productivity during the implementation phase. Compared with the number of tools for design and implementation, tools for use in the requirements phase are relatively few in number. We believe that as systems become more complex, the need to accurately capture the system requirements far outweighs the need to increase the efficiency of the design process.

This paper presents various aspects of the Ada Object Oriented Approach (AOOA) project currently being conducted by Harris. The objective of this project is to prototype various facets of a Requirements Specification Language (RSL) addressing the problems associated with current RSLs. This entails the development of an RSL based upon a graphical object-oriented format.

The description of a problem often identifies a large numbers of entities, their characteristics, behavior, and interactions. In order to clearly represent this information, AOOA emphasizes graphics as the most universal means of communication. This capability is supported in the form of three tools: an object editor, interface editor, and scenario editor. As the requirement engineer expresses his concept using the toolset, the resulting information is automatically stored in a Problem Specification Database (PSD).

The information stored in the PSD is gathered during all phases of the development lifecycle, from concept exploration through design and implementation to maintenance. From this foundation, the design team can use the AOOA toolset to elaborate the requirements into a design and implementation. Note that once the functional requirements have been elaborated, the resulting information bears a strong resemblance to an Ada program. Thus, the AOOA toolset contains an Ada source code generator to transform the information contained in the PSD into corresponding Ada source code. This method of generating Ada source code allows a project to be maintained by modifying the specification instead of the code. When faced with changing requirements, the requirements engineer can modify the information contained in the PSD. A new implementation may then be generated, reusing those portions of the specification that did not change.

The author received a BS in Computer Science from Louisiana Tech University in 1981. Since joining Harris, the author has been working as the principal investigator of the Ada Object Oriented Approach. As the principal investigator, the author was responsible for developing the concept, as well as initiating an internal research project from which the initial set of tools were created.

N89-16330

TOWARDS A GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT METHODOLOGY

Ed Seidewitz / Code 554
Mike Stark / Code 552
Goddard Space Flight Center
Greenbelt, MD 20771

1. INTRODUCTION

An _object_ is an abstract software model of a problem domain entity. Objects are packages of both data _and_ operations on that data [Goldberg 83, Booch 83]. The Ada (tm) package construct is representative of this general notion of an object. _Object-oriented design_ is the technique of using objects as the basic unit of modularity in system design. The Software Engineering Laboratory at the Goddard Space Flight Center is currently involved in a pilot project to develop a flight dynamics simulator in Ada (approximately 40,000 statements) using object-oriented methods. Several authors have applied object-oriented concepts to Ada (e.g., [Booch 83, Cherry 85b]). In our experience we have found these methodologies limited [Nelson 86]. As a result we have synthesized a more general approach which allows a designer to apply powerful, object-oriented principles to a wide range of applications and at all stages of design. The present paper provides an overview of our approach. Further, we also consider how object-oriented design fits into the overall software life-cycle.

2. OBJECTS AND OBJECT DIAGRAMS

We can model a _procedure_ as a mathematical function. That is, given a certain set of inputs (arguments and global data), a procedure always produces the _same set of outputs_ (results and global updates). A procedure, for example, cannot directly model an address book, because an address book has _memory_ (a set of addresses) which can be accessed and updated. Normally, the solution to this is to place such memory in global variables.

Figure 1 gives a representation of the above situation. This diagram uses a notation similar to [Yourdon 79] to show both data and control flow. The arrow from CALLER to PROCEDURE indicates that CALLER transfers control to PROCEDURE. Note that there is an implicit _return of control_ when PROCEDURE finishes. The smaller arrows in figure 1 show the data flows, which may go in either direction along the control arrow. Also, figure 1 includes an explicit symbol for the GLOBAL DATA. Control arrows directed towards this symbol denote data access, even though control never really flows into the data, of course. This convention indicates that the data is always passive and never _initiates_ any action.

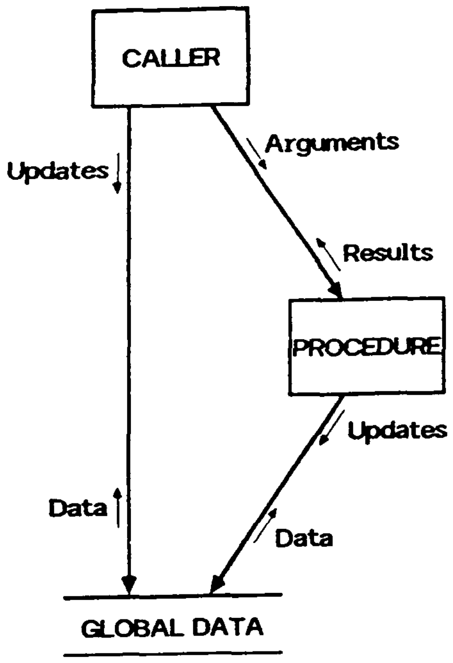


FIGURE 1 A procedure call

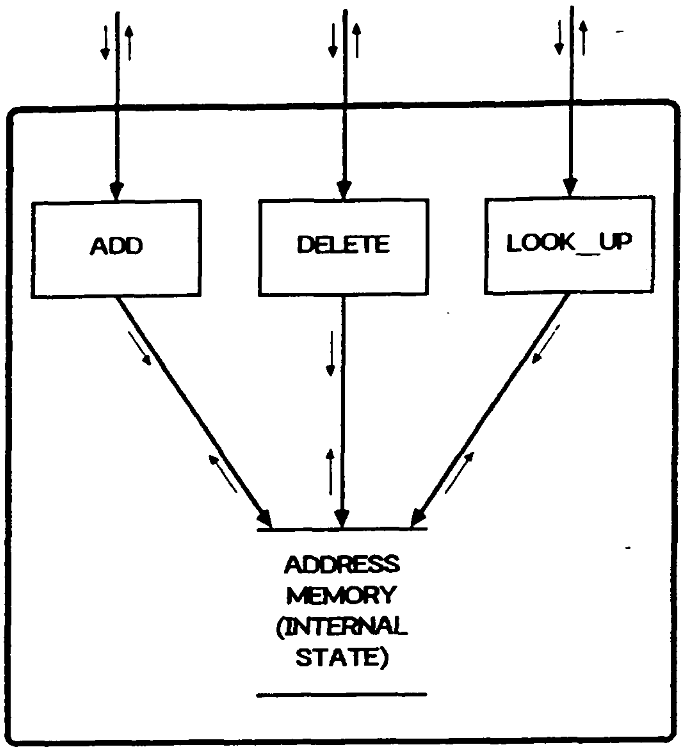


FIGURE 2 An ADDRESS BOOK object

The use of global storage leaves data open to illicit modification. To avoid this, an object packages some memory together with all allowable operations on it. We can model an object as a mathematical "state machine" with some internal state which can be accessed and modified by a limited number of mathematical functions. We thus implement an object as a packaged set of procedures and internal data, as shown in figure 2. For an address book object, the internal memory would be a set of addresses, and the allowable operations would be accessing an address by name, adding an address, etc. Unlike a procedure, the <u>same</u> arguments to an object operation may produce <u>different</u> results at different times, depending on the hidden internal state. We will diagram an object showing only its operational connections to other objects, as in the <u>object diagram</u> of figure 3 [Seidewitz 85a].

When there are several control paths on a complicated object diagram, it rapidly becomes cumbersome to show data flows or all individual procedure control flows. Therefore, an arrow between objects on an object diagram indicates that one object invokes <u>one or more</u> of the operations provided by another object and is <u>not marked</u> with data flow arrows. <u>Object descriptions</u> for each object on a diagram provide details of the data flow. An object description includes a list of all operations provided by an object and, for each arrow leaving the object, a list of operations used from another object. For example, the object description for DATE BOOK from figure 3 is:

```
    Provides:
      Next_Appointment () NAME + ADDRESS
      Get_Appointment (DATE + TIME) NAME + ADDRESS
      Make_Appointment (DATE + TIME + NAME)
      Cancel_Appointment (DATE + TIME)

    Uses:

      ADDRESS BOOK
        Look_Up

      CLOCK
        Get_Date
        Get_Time
```

Data in parentheses are arguments which flow along the control arrow, while unparenthesized data are results which are returned.

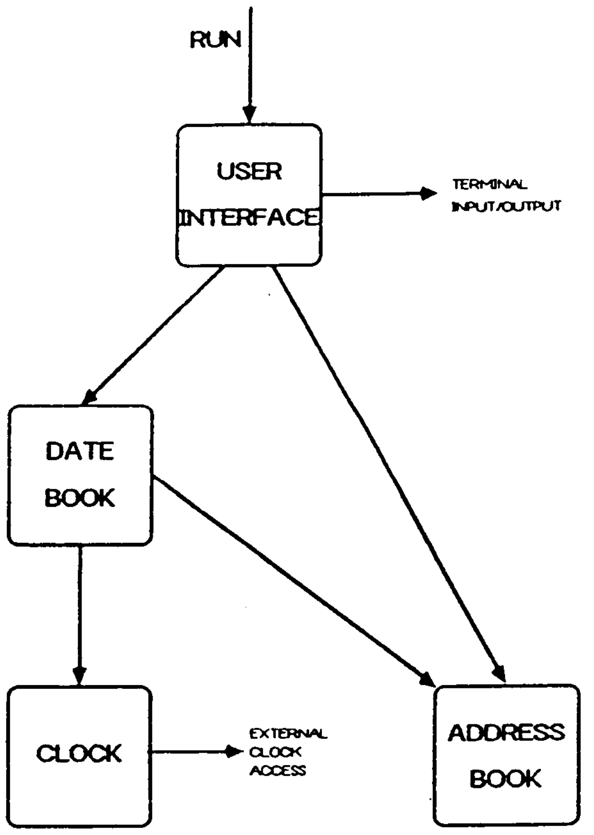


FIGURE 3 A simple schedule organizer

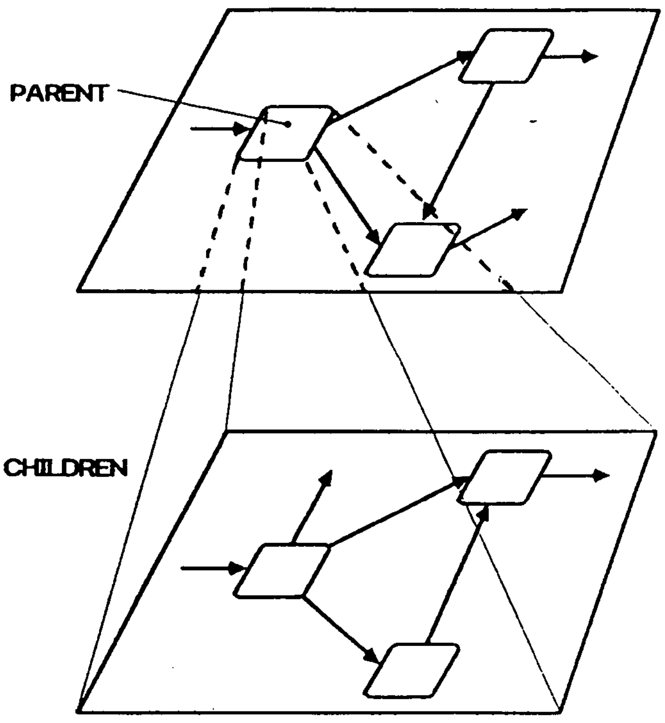


FIGURE 4 Parent-child hierarchy

3. OBJECT-ORIENTED DESIGN

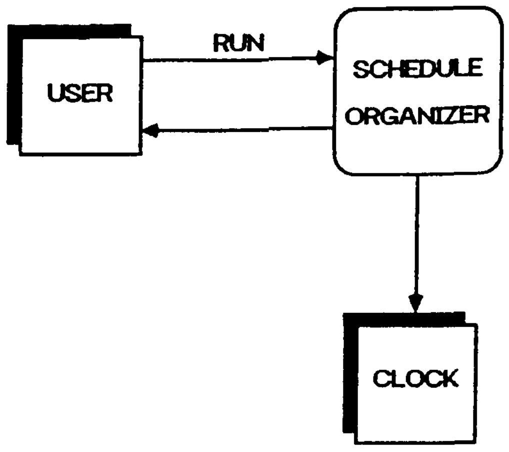
The intent of an object is to represent a problem domain entity. The concept of abstraction deals with how an object presents this representation to other objects [Dijkstra 68, Liskov 74, Booch 83]. There is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence. The following are some points in this scale:

Best $\downarrow$	Entity Abstraction - An object represents a useful model of a problem domain entity.
	Action Abstraction - An object provides a generalized set of operations which all perform the same kind of function.
	Virtual Machine Abstraction - An object groups together operations which are all used by some superior level of control or all use some junior level set of operations.
Worst	Coincidental "Abstraction" - An object packages a set of operations which have no relation to each other.

The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of information hiding states that such details should be kept secret from other objects [Parnas 72, Booch 83], so as to better preserve the abstraction modeled by the object.

The principles of abstraction and information hiding provide the main guides for creating "good" objects. These objects must then be connected together to form an object-oriented design [Seidewitz 85b]. Following [Rajlich 85], we consider two orthogonal hierarchies in software system designs. The parent-child hierarchy deals with the decomposition of larger objects into smaller component objects. The seniority hierarchy deals with the organization of a set of objects into "layers". Each layer defines a virtual machine which provides services to senior layers [Dijkstra 68]. A major strength of object diagrams is that they can distinctly represent these hierarchies.

The parent-child hierarchy is directly expressed by leveling object diagrams (see figure 4). At its top level, any complete system may be represented by a single object. For example, figure 5 shows a diagram of the complete SCHEDULE ORGANIZER of the last section. The object SCHEDULE ORGANIZER represents the "parent" of the complete object diagram of figure 3. The boxes labeled "USER" and "CLOCK" are external entities, objects which are not included in the system, but which communicates with the top level system object. Note the arrow labeled "RUN". By convention, RUN is the operation used to initially invoke the entire system.



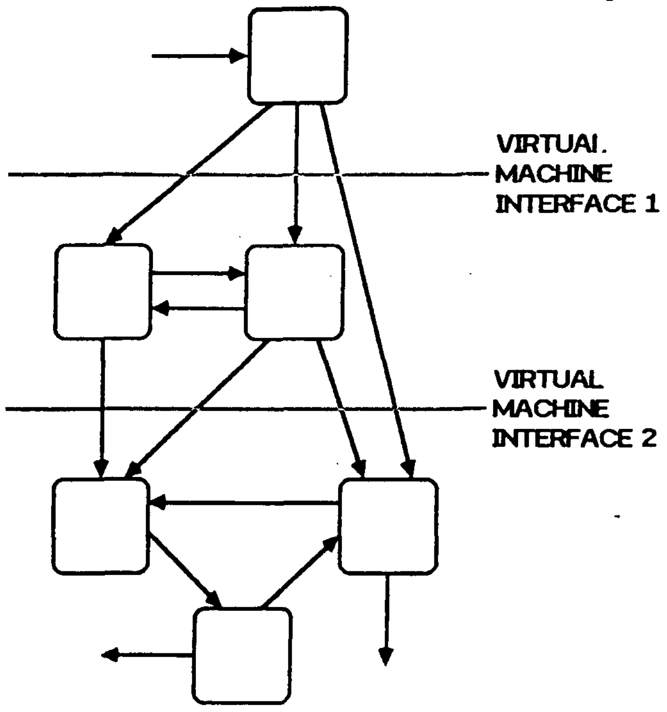


FIGURE 5 External Entities Diagram **FIGURE 6 Seniority hierarchy**

Figure 3 is the <u>decomposition</u> of the SCHEDULE ORGANIZER of figure 5. Beginning at the system level, each object can be refined in this way into a lower level object diagram. The result is a leveled set of object diagrams which completely describe the structure of a system. At the lowest level, objects are completely decomposed into <u>primitive</u> <u>objects</u>, procedures and internal state data stores, resulting in diagrams similar to figure 2.

The seniority hierarchy is expressed by the topology of connections on a single object diagram (see figure 6). Any layer in a seniority hierarchy can call on any operation in junior layers, but <u>never</u> any operation in a senior layer. Thus, all cyclic relationships between objects must be contained within a virtual machine layer. Object diagrams are drawn with the seniority hierarchy shown vertically. Each senior object can be designed as if the operations provided by junior layers were "primitive operations" in an extended language. Each virtual machine layer will generally contain several objects, each designed according to the principles of abstraction and information hiding.

The main advantage of a seniority hierarchy is that it reduces the coupling between objects. This is because all objects in one virtual machine layer need to know nothing about senior layers. Further, the centralization of the procedural and data flow control in senior objects can make a system easier to understand and modify. However, this very centralization can cause a messy bottleneck. In such cases, the complexity of senior levels can be traded off against the coupling of junior levels. The important point is that the strength of the seniority hierarchy in a design can be chosen from a spectrum of possibilities, with the best design generally lying between the extremes. This gives the designer great power and flexibility in adapting system designs to specific applications.

In the simple automated plant simulation system shown in figure 7, the junior level components do not interact directly. This design is somewhat like an object-oriented version of the structured designs of [Yourdon 79]. We can remove the data flow control from the senior object and let the junior objects pass data directly between themselves, using operations within the virtual machine layer (see figure 8). The senior object has been reduced to simply activating various operations in the virtual machine layer, with very little data flow. We can even remove the senior object completely by distributing control among the junior level objects (see figure 9). The splitting of the RUN control arrow in figure 11 means that the three objects are activated simultaneously and that they run concurrently. The seniority hierarchy has collapsed, leaving a homologous or non-hierarchical design [Yourdon 79] (no seniority hierarchy, that is; the parent-child hierarchy still remains). A design which is homologous at all decomposition levels is very similar to what would be produced by the PAMELA (tm) methodology of [Cherry 85a, Cherry 85b].

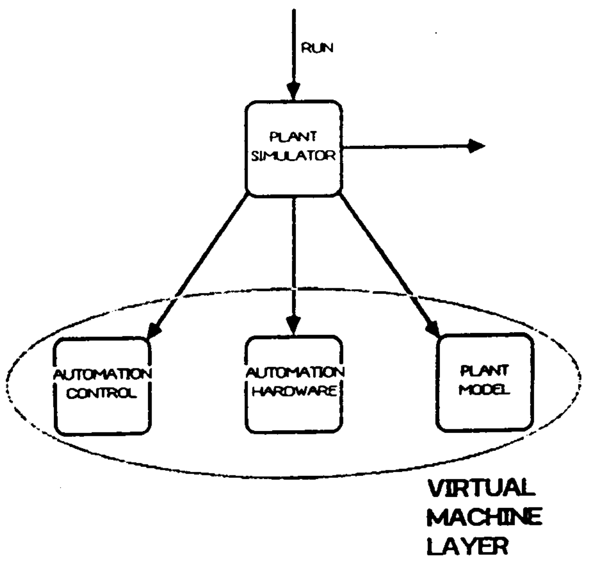


FIGURE 7 A simple plant automation simulation system

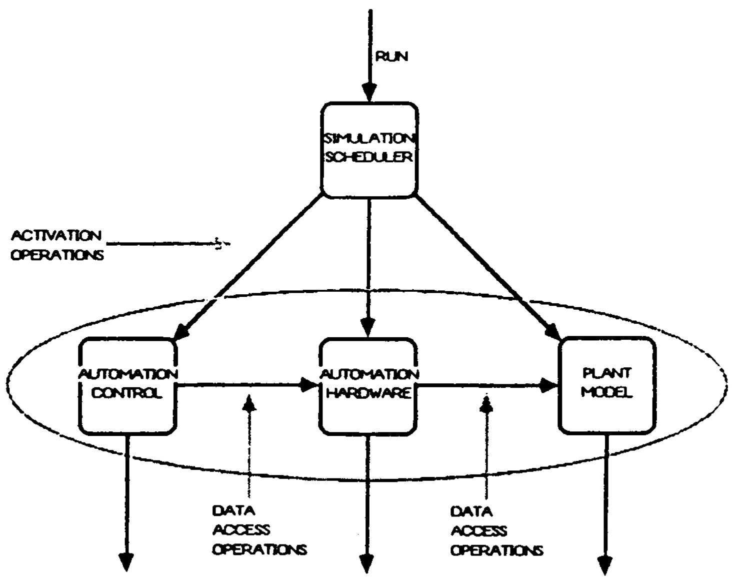
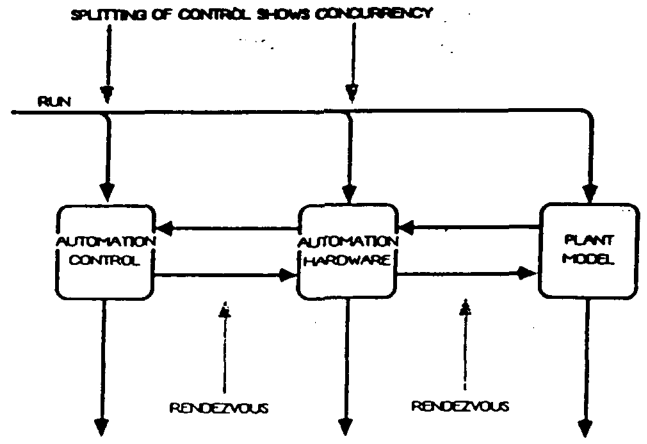


FIGURE 8 plant simulator with junior-level connections



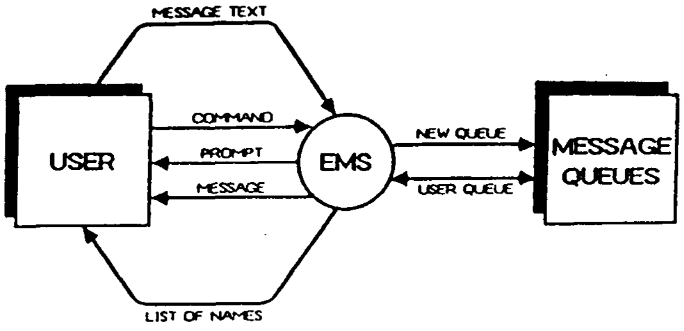


FIGURE 10 EMS context diagram

FIGURE 9 Plant simulator, homologous design

4. OBJECT-ORIENTED LIFE CYCLE

Object diagrams and the object-oriented design concepts discussed above can be used as part of an object-oriented life cycle. To do this, we must show that a specification can be translated into object diagrams, and that object diagrams map readily into Ada. We use structured analysis for developing the specification [DeMarco 79]. The data flow diagrams of a structured specification provide a leveled, graphical notation containing the information needed to represent abstract entities, but in a form emphasizing data flow and data transformation.

Abstraction analysis is the process of making a transition from a structured specification to an object-oriented design [Stark 86]. We will use a simplified version of an Electronic Message System (EMS) as an example of abstraction analysis. Figure 10 is the context diagram for EMS, and Figure 11 is the level 0 data flow diagram. EMS must allow the user to send, read, and respond to messages, to obtain a directory of valid users to which messages can be sent, and to add and delete users from that directory.

The first step of abstraction analysis is to find a _central entity_. This is the entity that represents the best abstraction for what the system does or models. The central entity is identified in a similar way to transform analysis [Yourdon 79], but instead of searching for where incoming and outgoing _data flows_ are most abstract we look for a set of _processes_ and _data stores_ that are most abstract. It may sometimes be necessary to look at lower level data flow diagrams to find the central entity. EMS is a system serving a person sitting at a terminal sending and receiving messages. On figure 11 we have circled the "current user" data store and the process 1.0 GET EMS COMMAND. Together this process and data store represent the user entering commands at a terminal. Thus they represent the central entity.

Next, we need to find entities that directly support the central entity. We do this by following data flows away from the central entity and grouping processes and data stores into abstract entities. In our example the USER DIRECTORY data store and the three processes (2.0, 4.0 and 5.0) supporting it form an entity. The process 3.0 ACCESS QUEUES with the data store USER QUEUE INDEX also form an entity. All these entities are circled and labeled on figure 11. We continue to follow the data flows and to identify entities until all the processes and data stores are associated with an entity.

Figure 12 is the entity graph for EMS. Squares represent entities, lines with arrows represent flow of control from one entity to another, and lines with no arrowhead represent interactions where flow of control is not yet determined. A "most senior" entity is placed into the design to give an initial flow of control. In the EMS example, entity EMS is this most senior object, and we have the USER INTERFACE entity "controlling" the external entity USER. This flow of control

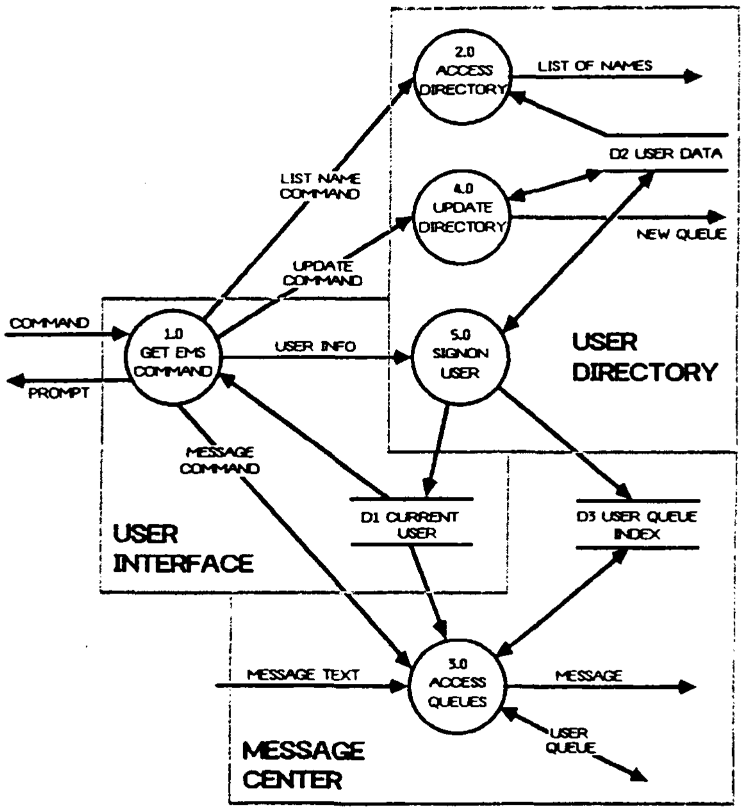


FIGURE 11 EMS level 0 data flow diagram

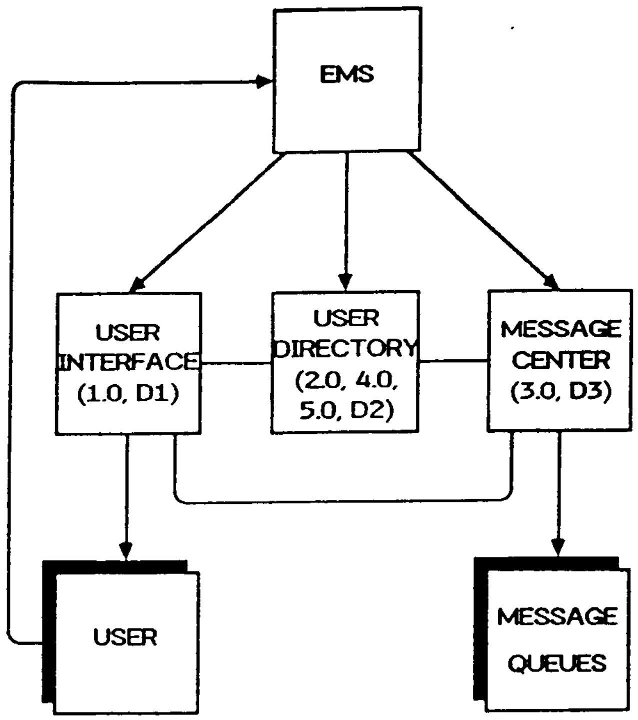


FIGURE 12 EMS entity graph

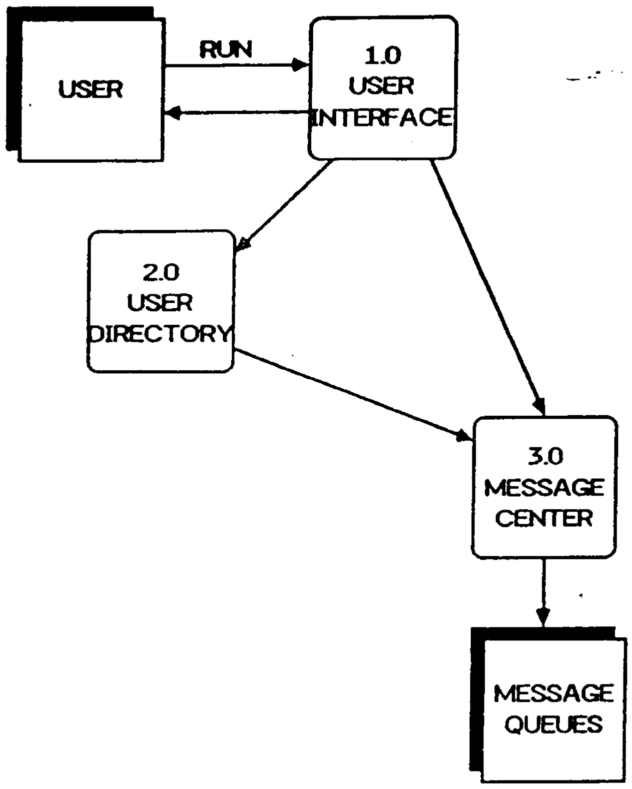


FIGURE 13 EMS object diagram

into USER will ultimately be implemented as read and write operations. Note also that the USER entity controls EMS. This flow of control represents the user invoking the EMS system. After this invocation control resides with EMS until the system is exited. All other potential interfaces are shown by lines with no arrows. The numbers inside the squares represent the processes and the data stores contained in the entity. This provides traceability from requirements to design.

The entity graph is the starting point for object identification. It shows entities with the highest abstraction possible and also shows all the possible interconnections between the entities. Since we are trying to balance design complexity, object abstraction, and control hierarchy, we will alter the entity graph to form the final object diagram. In EMS the entities are easily mapped into objects. The entities USER, USER INTERFACE, and EMS form a cyclic graph and therefore are on the same virtual machine level. We cannot combine an external entity into an object, but combining EMS and USER INTERFACE yields a single object that is senior to USER DIRECTORY and MESSAGE CENTER. Combining the two junior objects would simplify the design, but at the expense of abstraction, as the message passing mechanisms have little to do with the directory. We have also chosen to make USER DIRECTORY senior to MESSAGE CENTER, since the data flows are from USER DIRECTORY into data stores contained by MESSAGE CENTER. Figure 13 shows the resulting object diagram.

Needless to say, identifying objects is not always this simple. Usually there is a trade-off made between level of abstraction and design complexity, or a balancing of these two considerations and the virtual machine hierarchy. When these situations occur it is still the designer's judgement that must determine which side of the trade-off matters more for the application being designed.

Once the object diagrams are drawn we can identify the operations provided and used by each object. In the case of 2.0 USER DIRECTORY the operations are identified by examining the primitive processes contained within processes 2.0, 4.0 and 5.0 on figure 11. The data exchanged are identified by looking at data flows crossing the object boundaries, with the detailed information about the data being found in the data dictionary. The object description is produced by matching the operations and the data. The description generated for 2.0 USER DIRECTORY is as follows:

```
Provides:
  List_Names () LIST_OF_NAMES
  Add_User (USER_NAME + PASSWORD)
  Delete_User (USER_NAME)
  Signon (USER_NAME + PASSWORD) VALIDITY_FLAG

Uses:
  3.0 MESSAGE QUEUES
      Reset_Queue
      Create_New_Queue
```

Using the subset data flow diagram of processes and data stores that an object contains, the process of object identification can be repeated to produce a child object diagram. The only difference is that entities are identified based on how they support the object's operations, not by finding a central entity. This process is used until the lowest level of data flow diagrams is exhausted.

The transition from an object diagram to Ada is straightforward. The relationship between object diagram notations and Ada language features is:

Object Diagram	Ada Construct
Object	Package
Procedure	Subprogram
State	Package or task variables
Arrow	Procedure/function/entry call
Actor	Entries/Accepts (not covered in this paper)

Package specifications are derived from the list of operations provided by an object. For the EMS USER DIRECTORY object the package specification is:

```
package User_Directory is

  subtype USER_NAME is STRING(1..20);
  subtype PASSWORD is STRING(1..6);
  type LIST_OF_NAMES is array (POSITIVE range <>) of USER_NAME;

  procedure Signon (User: in USER_NAME; PW : in PASSWORD;
    Valid_User : out Boolean);
  procedure Add_User (U: in USER_NAME; PW : in PASSWORD);
  procedure Delete_User (U: in USER_NAME);
  function List_Names return LIST_OF_NAMES;

end User_Directory;
```

The package specifications derived from the level 0 object diagram are placed in the declarative part of the top level Ada procedure as follows:

```
procedure EMS is
  package User_Interface is
    procedure Start;
    ...
  end User_Interface;

  package User_Directory is
    ...
  end User_Directory;

  package Message_Queues is
    ...
  end Message_Queues;

  package body User_Interface is separate;
  package body User_Directory is separate;
  package body Message_Queues is separate;

begin
  User_Interface.Start;
end EMS;
```

For lower level object diagrams the mapping is similar, with package specifications being nested in the package body of the parent object. States are mapped into package body variables. This direct mapping produces a highly nested program structure. To implement the same object diagram with library units would require the addition of a package to contain data types used by two or more objects. This added package would serve as a global data dictionary.

The process of transforming object diagrams to Ada is followed down all the child object diagrams until we are at the level of implementing individual subprograms. If the mapping is done without explicitly creating library units the lowest level subprograms will all be implemented as subunits, rather than by embedding the code in package bodies.

5. EVALUATION OF THE METHODOLOGY

To measure how well abstraction analysis works as a methodology we must first define our criteria for a good methodology. We will use Barry Boehm's "Seven Principles of Software Engineering" [Boehm 76] as a basis of comparison. These principles are:

Manage using a sequential life cycle plan
Maintain disciplined product control
Perform continuous validation
Use enhanced top down structured design
Maintain clear accountability for results
Use better and fewer people
Maintain a commitment to improve the process

Abstraction analysis supports all these principles. The life cycle plan is supported by providing the abstraction analysis method for producing object diagrams, which are in turn mappable into Ada. This also provides a means of disciplined product control by tracing how Ada software implements an object oriented design, and also tracing how the design meets the specification. This traceability allows a manager to see that software meets its specification, and allows maintenance of specifications, design, and software to be consistent. Grady Booch's [Booch 83] work influenced our methodology, but did not provide a sufficient means of specifying large systems. Another drawback is that Booch does not define a formal mapping from a specification to a design.

The graphic notation supports a top down approach to software development. The leveling of both dataflow diagrams and of object diagrams allows the designer to start at a high level and work top-down to a design solution. The use of graphics also supports continuous validation by making design walkthroughs and iterative changes easier tasks to perform. Both Booch and Cherry [Cherry 85b] use graphics, but Booch's notation was not designed for large applications, and Cherry's methodology stops graphing after all the concurrent objects have been identified. The graphics used by structured analysis [DeMarco 79] provide the best analogy to how graphics are used in the object diagram notation.

The life cycle model we have defined also supports the remaining three principles. Objects are defined in the design phase and implemented as separate Ada compilation units. Tools such as unit development folders can be used to maintain accountability for completion of the design, implementation, and testing of objects. It is hoped that the object-oriented approach and the use of Ada will enhance both productivity and software reliability. This assertion will be tested by measuring the outcome of the pilot project in the Software Engineering Laboratory at Goddard Space Flight Center. The success of this methodology would allow better and fewer people to concentrate more effort on producing a good design.

Finally, we are certainly committed to improving the process. The object diagram notation and abstraction analysis have already seen much change since the initial versions were defined. Further refinement will be to define criteria for using parallelism, criteria for choosing between library units and the nested approach defined above, and to generate object-oriented approaches to software specifications and software testing.

6. CONCLUSION

Object diagrams have been used to design a 5000 statement team trainging exercise and to design the entire dynamics simulator. They are also being used to design another 50,000 statement Ada system and a personnal computer based system that will be written in Modula II. Our design methodology evolved out of these experiences as well as the limitations of other methods we studied. Object diagrams, abstraction analysis and associated principles provide a unified framework which encompasses concepts from [Yourdon 79], [Booch 83] and [Cherry 85b]. This general object-oriented approach handles high level system design, possibly with concurrency, through object-oriented decomposition down to a completely functional level. We are currently studying how object-oriented concepts can be used in other phases of the software life-cycle, such as specification and testing. When complete, this synthesis should produce a truly general object-oriented development methodology.

TRADEMARKS

Ada is a trademark of the US Government (Ada Joint Program Office).

PAMELA is a trademark of George W. Cherry.

REFERENCES

[Boehm 76]	Boehm, Barry W. "Seven Basic Principles of Software Engineering," NASA/GSFC Engineering Colloquium, 1976.
[Booch 83]	Grady Booch. Software Engineering with Ada, Benjamin/Cummings, 1983.
[Cherry 85a]	George W. Cherry. PAMELA: Process Abstraction Method for Embedded Large Applications, Course notes, Thought**Tools, January 1985.
[Cherry 85b]	George W. Cherry and Grad S. Crawford. The PAMELA (tm) Methodology, November 1985.
[DeMarco 79]	Tom DeMarco. Structured Analysis and System Specification, Prentice-Hall, 1979.

[Dijkstra 68] Edsgar W. Dijkstra. "The Structure of the 'THE' Multiprogramming System," Communications of the ACM, May 1968.

[Goldberg 83] Adele Goldberg and David Robison. Smalltalk 80: The Language and Its Implementation. Addison-Wesley, 1983.

[Liskov 74] Barbara H. Liskov and S. N. Zilles. "Programming with Abstract Data Types," Proc. of the ACM Symp. on Very High Level Languages, SIGPLAN Notices, April 1974.

[Nelson 86] Robert W. Nelson. "NASA Ada Experiment -- Attitude Dynamic Simulator," Proc. of the Washington Ada Symposium, March, 1986.

[Parnas 72] David L. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, December 1972.

[Rajlich 85] Vaclav Rajlich. "Paradigms for Design and Implementation in Ada," Communications of the ACM, July 1985.

[Seidewitz 85a] Ed Seidewitz. Object Diagrams, unpublished GSFC report, May 1985.

[Seidewitz 85b] Ed Seidewitz. Some Principles of Object Oriented Design, unpublished GSFC report, August 1985.

[Stark 86] Mike Stark. Abstraction Analysis: From Structured Specification to Object-Oriented Design, unpublished GSFC report, April 1986.

[Yourdon 79] Edward Yourdon and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979.

SESSION D.5	CAIS PANEL
Panel Chair:	David Pruett NASA Johnson Space Center
Panel members:	
	Clyde Roby Jack Krammer Institute for Defense Analysis Alexandria, Virginia
	Sue LeGrand SofTech Houston, Texas
	Robert Stevenson Gould Electronics Fort Lauderdale, Florida
	Robert Fainter Virginia Tech Blacksburg, Virginia
	Hal Hart TRW Defense Systems Group Redondo Beach, California

N89-16331

SOME DESIGN CONSTRAINTS REQUIRED FOR THE ASSEMBLY OF SOFTWARE COMPONENTS: THE INCORPORATION OF ATOMIC ABSTRACT TYPES INTO GENERICALLY STRUCTURED ABSTRACT TYPES

Charles S. Johnson

ABSTRACT

It is nearly axiomatic, that to take the greatest advantage of the useful features available in a development system, and to avoid the negative interactions of those features, requires the exercise of a design methodology which constrains their use. A major design support feature of the Ada language is abstraction: for data, functions, processes, resources and system elements in general. Atomic abstract types can be created in packages defining those private types and all of the overloaded operators, functions and hidden data required for their use in an application. Generically structured abstract types can be created in generic packages defining those structured private types (i.e. lists, trees), as buildups from the user-defined data types which are input as parameters. A study is made of the design constraints required for software incorporating either atomic or generically structured abstract types, if the integration of software components based on them is to be subsequently performed. The impact of these techniques on the reusability of software and the creation of project-specific software support environments is also discussed.

INTRODUCTION

The reusability of Ada software developed in support environments will be wholly dependent upon the quality of those environments. The ability of programmers that are relatively inexperienced in Ada to generate reusable software will be enhanced by an environment rich in already reusable software components, which act as models for good design. In an analogy to a factory, components which are tooled to fit can be easily assembled. Atomic abstract types define objects which represent the discrete phenomena that are the subjects of the system development. Generically structured abstract types organize the objects of the system in a manner representing the relationships between those objects. If atomic and generically structured abstract types are defined according to some general design goals and constraints, then the subsequent assembly of these software components is made considerably easier.

BRIEF BACKGROUND

Kennedy Space Center/ Engineering Development/ Digital Electronics Engineering Division is in the process of prototyping distributed systems supporting I & T applications, particularly the Space Station Operations Language (SSOL) System, which is the I & T subset of the User Interface Language

(UIL) for the Space Station. The discussions in this paper were developed from the results of systems designed and developed in Ada to demonstrate the general feasibility of creating software support environments which maximized the reusability of software components. The Ada environment used was that of VAX Ada under VAX/VMS.

OBJECT DEFINITION IN ADA

The design and development of software components that meet the needs of the user community can be viewed largely as an effort to define and refine the definition of abstract objects and their associated operations in computer systems. The definition of objects in these systems is akin to a simulation effort. There is a direct correlation between the effectiveness of programs and the fidelity with which objects in those programs simulate the behavior of the external phenomena they are intended to represent. For example, an element in a scheduler queue, representing a process awaiting execution, must reflect the correct state of the process (priority, blocked for I/O, etc.) for the scheduler to function properly. The element must be distinguishable from other elements and not lose identity or integrity during operations.

As in simulation efforts, the goals and objectives for defining an object in a system should be specified at the outset. The system functional requirements should drive the process, while the scope of the system concept constrains development to areas that are productive.

SIMPLE TYPES

An object is characterized by it's attributes and the operations which mediate it's interactions with other objects in the environment. In the Ada language, the process of object definition begins with selection of base type or the creation of a composite type.

Objects whose behavior is simple enough to be modeled by a numeric value, can be represented by subtypes or derived types, of numeric or discrete types. The subtype definitions can include range constraints, and in the case of non-discrete numeric types (UNIVERSAL_REAL, UNIVERSAL_FIXED), they can include limits of precision for the representation. Declared objects of subtypes are, however, compatible with their base type and subtypes of the base type, which can allow erroneous combinations by operations allowable in the base type (adding MINUTES to HOURS, for example).

If the allowable operations of these base types are unsuitable, they can be restricted by the use of a derived type, which inherits the operations of the base type, but only for declared objects of the derived type (incompatible with the base type). This can yield dimensional errors, however, for multiplies and divides of objects of the same type (FT * FT = FT, instead of FT squared). In these cases subtypes and derived types are too simple in behavior to correctly represent the objects in applications, and composite types must be used. [1]

COMPOSITE TYPES

Objects which are characterized by collections of components or attributes are defined in Ada by the use of composite types: arrays and records or by access types which designate composite types. Objects which are collections of compatible components are represented by arrays, whereas objects which have various kinds of attributes are represented by records or access types designating records. Objects with attributes, and which have complex interactions with other objects in the system, would seem to be the more useful, although these are the most complex to define.

Objects with attributes interact with each other by the means of those attributes, under the control of the allowable operations of the objects. These interactions can produce modifications and deletions of the objects or creation of new and different kinds of objects. In Ada, the operations are defined as subprograms (functions and procedures) with parameters of the object type or subtype.

The operations which correspond to functions can be overloaded onto the set of computer math symbols for the given types. A function producing a scalar dot product from two input vectors could be given the name "*", for instance. At the same time, a function producing a vector cross product from two vectors, could also be named "*". The compiler would resolve these two operations from the type of the returned object. The compiler cannot, however, resolve these operations when the type of return is unknown. Vector products defined in this way could not then be embedded in longer equations, where they would generate intermediate results of indeterminate type.

DIFFERENTIATION

There are different levels of definition for a system, it's objects and operations. Definition of the gross structure of a system can typically be generated, in a fairly simple manner, by the object-oriented or functional decomposition methods. Definition of the fine structure of the system involves different methods, which produce results of greater complexity. One proposed second-stage method is differentiation.

If the definition of an object is found to be too amorphous to yield the correct behavior, differentiation can produce separate and more distinct types of object. The differentiated types will tend to be closely coupled and capable of interacting with the same operations that the undifferentiated type allowed for interactions of objects of that type. Where they differ in behavior is that area of operations or attributes that required the split. This type of tightly coupled interaction between different types is produced automatically in subtypes of the same base type, through inheritance. Subtypes, however, are very tightly coupled, and can only differ from each other in terms of ranges (numeric or discrete subtypes), numbers of components (constrained array subtypes) or discriminant values (constrained record subtypes).

If the differentiation is more extensive, requiring objects of differently structured base types, then all of the allowed interactions between the objects must be defined more laboriously. The rewards of this diligence, which are unique to Ada, are the isolation of system complexity to a package defining all the closely coupled interactions, while the programming using these types and operators can proceed at a higher level.

OBJECT LIFE CYCLE

Definition of a system down to the fine structure produces a definition that is no longer intuitive, and requires some non-intuitive method for it's verification. The life cycle of an object may prove to be useful in providing a path to follow, in the analysis of complex objects.

All objects have their own life cycle, however brief, in the system environment. They are created and deleted by an operation or system event, either explicitly or implicitly. During their life they interact with other system objects, with results dictated by the appropriate operations.

The verification of the results of object definition can be performed by a "walkthrough" of the object life cycle. During this process, the defined attributes and operations of the object can be evaluated in the light of the events it experiences: creation, interactions and demise. If, under these circumstances and within the scope of the requirements, the abstract object behaves similarly to the phenomena which it is intended to represent, then the object with it's attributes and operations can be expected to reliably support the development of applications concerning that phenomena.

The Ada language features which directly support the definition of objects are packages and private types. Packages contain the definition of the object and allowable operations, which are visible, and the implementation, which is hidden. Private types further close the window of visibility, allowing only higher-level or interface attributes of the object definition to be visible

TWO CLASSES OF ABSTRACT TYPES

For the purposes of the assembly of software components, there appear to be two broad classes of private types. The types which support the definition of objects as discussed above are called, only for the purpose of distinction, atomic abstract types. These types represent the discrete phenomena which are the subjects of system development, and are defined in packages as private types. They have the indivisible property of atoms, and can be incorporated into the second class of types: the generically structured abstract types.

Generically structured abstract types are managed by generic software components (packages or subprograms), and are built-up from application-defined types which are contained as components of the generic structure. These structured abstract types organize the objects of the system in a manner representing the relationships between objects, and they shall be discussed first.

GENERICALLY STRUCTURED ABSTRACT TYPES

These structures, built-up from application-defined atomic abstract types and managed by generic packages, support the basic organization of the elements of the system. The organization of objects in a structure is a representation of the relationships between those objects, which can be either static or dynamic in nature.

The specification of a generic package is parameter driven. The generic formal parameters of a generic package are the basis and controlling factor in the reusability of the package. The use of generic software has implications, however, for the design of atomic abstract types which are later to be used in an instantiation of that software. The benefits of reusability can only be fully realized if the design of atomic abstract types follows distinct lines.

Taking an example of a generic sorting routine, it can readily be seen that the reusability of the routine is dependent upon the initial typing of the generic formal parameters and the matching rule for generic formal parameters. If the parameter is typed as simply private, then the maximum reusability is achieved, because it will match nearly everything (except discriminant or limited private types). However, if the parameter is typed as a real (digits <>) or integer (range <>), the operations that are consistent with those types will be available to the internals of the generic, but at the expense of only allowing those types as parameters.

It should be noted here, that although a generic formal parameter of the limited private type would extend the generality of the generic software component, it is not useful due to the lack of both assignment and compare for equality within the generic. Without assignment, components of the structure cannot be set, or initialized to any value.

The concept of generic programming turns private types and visibility inside out. In the case of a generic package, the structure of a type passed as a formal parameter is not visible to the package which manipulates it.

In the support of generic structures, typically all that is needed is the assignment function ":=", the compare for equality function "=", and an ordering function ">". The assignment and compare functions are available with type private parameters, and the ordering function ">" can be passed as another formal parameter. With no other details or operations, structures like lists, queues, indexes, and hierarchical tree structures containing objects of the generic formal parameter type can be defined and maintained by the package.

ATOMIC ABSTRACT TYPES

The atomic abstract types are the components which fit into the generically-structured abstract types, during the assembly of software components. As such they must be crafted to fit easily into the generic structure.

As has been noted, generic formal parameters of the maximum range of applicability are those of the private type. The problem then is to design atomic abstract types that match the simple model of the private type: assignability and comparability.

Discriminant types, although very useful on their own for the development of objects with constraining attributes, are fairly disfficut to use in conjunction with generic software. Very quickly it is found that, to match a discriminant type with a generic formal parameter, the types for each individual constraint must first be passed as generic formal parameters. Then the discriminant type must be passed with it's constraints. Unconstrained types are not allowed. Generic formal parameters of this combination should be fairly difficult to match with any type other than the type initially matched, making for extremely reduced reusability.

Access types, which are the foundation of the dynamic structure of generically structured abstract types, are of little use in constructing atomic abstract types. They perform the assignability function more or less according to the simple model of private types, however they do not create a copy of the designated object (object pointed to), but instead copy the access object value (pointer address) onto the new object. This creates a shared object, with a certain loss of object identity, and could cause integrity problems inside the generic structure which incorporates the access object as a component.

The ordering function ">", used to order the elements of a generic structure (index, tree), can be defined by overloading the ">" function for the access object, to create a function comparing the designated objects values (for a string access type, the ">" would compare the designated strings).

The compare function "=" is another matter, however. It exists for access types, but compares the values of the access objects to see if they designate the same designated object. The "=" can only be overloaded if the abstract access type is declared as limited private instead of private. When this is done, however, the assignment operation ":=" is lost (and cannot be overloaded), which is needed for internally manipulating the generic structure inside the generic package.

Embedding the access type in a non-discriminant record would not change the reference nature of the contained object, and the problem of compares.

Embedding discriminant types, however, is very successful. As long as the constraint is not needed for data validity, this technique can hide the discriminant type within a non-discriminant record. The non-discriminant record will match a generic formal parameter of type private. This allows, for instance, a variable string (unconstrained array type), to be contained within a non-discriminant record, and passed to generic procedures easily.

DESIGN GOALS AND CONSTRAINTS FOR ATOMIC ABSTRACTION

In the process of feasibility prototyping for the generation of application independent software support environments, the following design goals and constraints were found to yield, for packages supporting atomic abstract types, the maximum in abstraction, flexibility, and potential for generic structure incorporation:

1. Package-defined atomic objects being declared in the application software should, where possible, be defined as abstract types, that is, made private.

2. If the operations of an object are analogous to those of standard objects already in the system, overload the same names for the operations. This enhances readability and learnability of the application software support environment. Do not, however, overload names with non-analogous functions.

3. The functions performed by the operations of an object should be intuitive. The action performed by an operation should be predictable from the context of the application software.

4. The outcome or result of operations of an object should be intuitive. The kind of object produced by operators, for example, should be predictable from the context of the application software.

5. Maximize the completeness of the application interface to the atomic type defined in the package. Give the application developer all of the operations required to manipulate and combine objects, in an easy-to-use yet well controlled manner.

6. Maximize the potential use of reusable software incorporating the abstract atomic type into generically structured types. This can be accomplished by defining types that perform simply under the operations of assignment and comparison (not discriminant types or access types, which follow a more complex model).

DESIGN GOALS AND CONSTRAINTS FOR GENERIC ABSTRACTION

In the process of feasibility prototyping for the generation of application independent software support environments, the following design goals and constraints were found to yield the maximum in reusability and flexibility for packages managing generically structured abstract types:

1. Package-managed generic objects that are declared in the application software should, where possible, be defined as abstract types, that is, made private.

2. Maximize the generality of the package. This comes from the use of formal generic parameters, particularly for types, that match the widest variety of application input types (type private instead of digits <>, for example).

3. Maximize the usability of the application interface to the package. Extend, as far as possible into the application domain, access to the structures managed in the package, without violating the integrity of the internals, or the independence of the application from the generic software component (generality).

4. Maximize the completeness of the application interface to the package. Give the application developer all the operations required to access and manipulate the internal structures, in a package-controlled manner.

5. Support, if possible, multiple objects with the same package. This limits the need to re-instantiate the package several times within the same scope, for processing of multiple objects.

6. Design for flexibility: a single tool, suited to a wide range of applications, is more likely to be remembered, and used by developers.

7. Cover the infrequent failure modes. Most failures of algorithms and processing logic in programs occur at the extremes of their domain of applicability. Testing should cover the ends of ranges and the infrequent states of the application. If the software component is reusable, it will be used in a wider range of applications, and the infrequent failure modes will occur more frequently.

PACKAGES SUPPORTING GENERICALLY STRUCTURED ABSTRACT TYPES

The index package, described as a list of elements ordered by another set of associated elements or keys, will be used as an example for a package supporting a generically structured abstract type. The index structure itself should be a private type. It should be defined in the package specification, not hidden, so that it can be declared as an object in the scope of the application. The package should be capable of accessing and managing several objects of type INDEX, so there should be a USE_INDEX function, which selects the appropriate object, and sets a package-internal access object to the same value as that passed as the USE_INDEX parameter. Then there will be two access objects pointing to the index structure internals, one in the application scope, one in the package scope.

Since the access object in the application scope cannot be changed, neither can the access object in the package scope (unless there is a subsequent USE_INDEX call). They must stay aligned. This means that the INDEX access object cannot designate the head of the index-list, but must instead designate an access object that designates the head of the index-list. This is in case an insert must be made at the head of the index, and the access object that designates it must be modified.

The importance of having the index object in the scope of the application is in the flexibility of use of the object at the application level. The developer should be capable of passing the object as a parameter to subprograms developed at the higher level. If the object of type INDEX is hidden, this flexibility is not there.

The indication of success or failure of an operation (add/delete, search, etc.) should be available for the application, for the purpose of logical tests and conditional branching. It should be contained in the package scope, visible in the package specification, and it can be called STATUS. Values contained in status can be defined in the package specification to show conditions (END_OF_LIST, ELEMENT_NOT_FOUND, etc.).

CURRENT-NODE POINTER FACILITY

One question about package operations that must be answered before the design phase is about the context-sensitivity of operations. Higher level operations, like those involved in command languages, are typically constrained to be context insensitive, on a line-by-line basis. This means that the interpreter of the command or function requires no information, other than that in the command, to interpret it completely. There is no contextual basis.

This can be effectively at a high level of application, but is difficult for the implementation of any complex functionality. For the package managing a complex structure, it is really necessary for the package to keep a contextual indication of the current position of the search through the structure in between calls. A USE_INDEX call to a new index would reset this position indicator, of course, as would any search, add/delete, or sequential positioning call. This prevents the need for a node search upon every call. This position indication variable can be called CURRENT.

CURRENT is of necessity an access object. If CURRENT is kept in the application scope, it must be passed in the subprogram interfaces of every operation. Also, being in the application scope, synchrony can be lost between USE_INDEX calls (pointing to the wrong INDEX designated structure).

If CURRENT is kept in the package, the package can track application context, and reset CURRENT upon USE_INDEX invocation. Also, it should be hidden, because it would be difficult for the application to interpret it anyway.

With these design issues decided, a generic package for managing INDEX objects can be developed.

REUSABILITY ISSUES

Reusability is generally discussed in terms of taking software written at other sites, and not necessarily on the same machine, and porting it for use in an application. There is a context here, which can be called inter-project reusability. This kind of reusability is based on two types of software development.

In the first type of reusable software, software components or interfaces to non-Ada components are produced for general application support areas, like DBMS, user interface software, graphics, communications, data reduction and others, even AI. These will certainly be necessary to include, as they are more expensive to develop than to buy. They will also be the most commercially available.

In the second type of reusable software, and with far less availability, software components are written targetting the application area of interest. These will probably be less of a fit to the specific application, with fewer packages to choose from.

In the I & T area, high performance software is hard to obtain, and will be in the future. This is due to the narrow market and the very high degree of system dependence of the applications developed. In application domains with parameters like those of I & T, the major gains in Ada reusability will be those derived from software designed and developed in the same project.

This kind of reusability can be called intra-project reusability, and comes from design by abstraction. High level software can be produced for specific application domains by the production of packages tailored to support those domains.

Packages implementing private types can be developed that support the objects and operations representing the phenomena which are the subject of system development. If these objects and operations simulate the behavior of those phenomena well (within the purposeful domain), then the applications developed using them will be higher level, and generally more effective and maintainable.

Generic packages can also be developed supporting the static and dynamic relationships between objects in the system. If these packages can be made flexible and with maximum reusability, then the objects of the system can be organized by instantiation of those packages, allowing the system relationships to be established on a high order level in a logical way.

The reuse of both sets of software can be enhanced by establishing design constraints on each, so that the software components of the system can be assembled with maximum likelihood. The design goals and constraints on Ada software can not be effectively left as an afterthought.

PROJECT-SPECIFIC SOFTWARE SUPPORT ENVIRONMENTS

The effectiveness and reusability of software generated by relatively inexperienced Ada programmers will be directly related to the project-specific software environment that exists when they first enter the project. It will always be found that it is easier, quicker and more reliable to construct anything from pre-fabricated components that fit together as well as Lego blocks do. Two things are required to build a good set of blocks.

First, the objects (the logical atoms and molecules of the system) and their operations must be represented well by packages supporting those atomic abstract types and all of their support functions. Secondly, the relations organizing the objects of the system must be supported with generic packages that are flexible and easy to use.

In the internals of both of these packages can be buried the hidden complexity of the system, and some of the system dependencies as well. In this way, technology insertion into the system can be accomplished directly, without negatively affecting the applications of the system. [2]

Finally, a good set of blocks is not sufficient to build a system. The builder has to know what he is building to be effective. There is no substitute for Requirements Analysis and Functional Decomposition using data flows and similar techniques to express what a system does in a manner traceable back to the original User Requirements. The development of Ada and the object-oriented design methodologies which Ada directly supports will eventually prove, however, to be a large step on the way to cracking the problem of what to do after the System Requirements are assigned to the top-level components of the system.

ACKNOWLEDGEMENT

I gratefully acknowledge the support given by the Kennedy Space Center/ Engineering Development/ Systems Integration Branch in supplying the computer facilities for the feasibility studies that provided the basis of this work. I also thank my wife, Bronwen Chandler, for her support.

REFERENCES

1. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station, F.4.4.

2. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station, B.4.3.

N89-16332

CERTIFICATION OF ADA PARTS FOR REUSE

Gregory A. Hansen*,
General Dynamics, Data Systems Division, San Diego, CA

S.D. Spaulding, General Dynamics, Data Systems Division, San Diego, CA

Glenn Edgar, General Dynamics, Data Systems Division, San Diego, CA

* Currently employed by the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA

Introduction

One of the claims made by proponents of Ada is that Ada software is highly reusable. The fact that specifications are compiled and accessible would make reusability seem easily achievable. However, specifications give only a limited amount of information about a package; moreover, a specification cannot help determine whether a package "worked", or how well it worked.

This problem has led to the concept of "certifying" Ada parts for reuse; that is, determining the worthiness of a part as a reusable component. This paper addresses issues that are critical to reuse: the characterization of part performance, design for reuse, and correct utilization of parts. The paper will then address current areas of study beneficial in the development of a certification process.

I. Characterization of Part Performance

Ada has two features which support reusability: specification compilation and the ability to declare instantiations of generic units. Ada specifications allow a part's interfaces to be defined, but are not sufficient for defining reuse. It is important to have information about a part's performance in a reuse operation to determine computational requirements, accuracy of calculations, etc.

Performance is not something that is easily quantified; however, attributes associated with performance are definable. A part's performance is definable by its behavior, or intended function, and the computational resources it extracts from the system when executing. These attributes are directly related, and not independent. They must be considered in the scope of both normal processing and exception handling.

The explicit separation of exception handling and normal processing is essential for modularization. Without exceptions, nested flags are required for error recovery management. This leads to the intermingling of error (i.e. exceptions) and normal processing, which leads to unmanageable code. Ada provides for the separation of exception handling and normal processing, and this separation is mandatory for parts reuse.

Exception handling consists of three steps: exception detection, correction, and recovery. These steps should be handled at different places in a software system. The exceptions that are raised, and the method of handling those exceptions, are not contained in a package specification. This information is essential to the part certification process.

How exceptions are handled determines the behavior of an Ada part and affects the performance of that part. The detection, correction, and recovery philosophy of a system has direct bearing on the computational requirements of that system, as does frequency of exceptions. Subjects such as recovery vs. restart and process synchronization must be addressed. Exception handling standardization could be an important factor in the certification process.

GENERICS.

Generic programming seems to provide a logical approach to the certification of reusable software. However, some obstacles to reusability, such as exception handling, still exist. Specifications for generics give no more information concerning the behavior or performance of the corresponding body than specifications of packages, subprograms, or tasks. Since Ada does not allow exceptions to be passed as parameters for the instantiation of generic parts, the use of a generic as a reusable part is somewhat constrained. The usual exception declaration interface between a system and a generic package is the package specification. Thus the identity and meaning of the exception is determined by the generic package, not the host program. This constitutes a reversal of accepted top down design techniques.

Another way of interfacing exceptions and reusable generic parts is to have both the system and the generic unit depend on a specifaction package of exception declarations. This technique would be consistent with top down methodologies, but would require a high degree of cooperation between system implementers and the designers of reusable parts.

A third technique of exception interfacing involves the implementation of subprograms which raise exceptions. The subprograms would be elaborated in the system's declarative environment and passed as actual parameters to generic instantiations. This technique solves the problem, but at the cost of efficiency, elegance, and design clarity.

CONSIDERATION OF TIMING

Another factor affecting certification of reusable parts is the hard timing requirements of a part. This information is not extractable from a package specification, and varies from system to system. In early computer architectures, timing was a fairly easily calculated quantity. However, multitasking software systems and new architectures which use cache, floating point accelerators, and other features, have direct influence on timing. In fact, non-package constraints such as context switch times have become as important as package timing itself.

Since parts can be viewed as tree structures with many branches; where exception handling and timing must be considered, the characterization of a part's performance and its certification are indeed very complex.

II. Constraints in Design for Parts Reuse

The design of software parts must be done in a context independent manner; that is, no assumptions should be made about input conditions. All possible error conditions should be anticipated and treated as exceptions. The exception handling implementation must be explicitly documented.

The Ada compiler run time default error checking features should not be used, except as a redundant check. If run time error checking is turned off for speed reasons, then flaws potentially exist in the system. Therefore, error conditions must be handled by the package. This philosophy, unfortunately, can lead to speed impacts within the system.

If there are time constraints placed upon a part, then a "cost" analysis must be performed on that part prior to its implementation, and the results of that analysis must be captured for later use. A hierarchical funtional decomposition methodology, such as data flow or Petri nets, can be used in the analysis process. As will be discussed later, expert system technology can be applied to the performance of the "cost" analysis.

It should also be mentioned that there exists a potentially large number of specific coding and design practices that can adversely impact reusability at both the system and part level. To fully identify these practices and address their relative impact will take time and experience, and such a discussion is beyond the scope of this paper.

III. Certification and Reuse

Ideally, a certified "part" should be a reusable part. However, it is probable that parts that are considered to be 100% certified are going to be small segments of code with limited application. The process of certifying large segments of code is extremely complex.

This paper has made several points concerning the reuse of Ada parts:

- o Ada specification packages are insufficient for determining reuse
- o Behavior and performance of a part must be explicitly defined and extractable
- o Exception handling is an important factor in both behavior and performance
- o Generics offer a logical approach to certification of reusable parts but have certain constraints
- o Hard timing requirements must be stated, and are subject to variations created by hardware and software environments
- o Run time implementations must be considered as influencing a part's behavior

Artificial Intelligence can provide some technology to reduce the complexity of analysis for reuse. In particular, expert system technology and object-oriented design can be applied to the problem. Object-oriented design is a term used to define a methodolgy of software development in which data items in a software system are defined in terms of their attributes, as well as in terms of their relationship to other data items in the system. Object-orientation has led to the concept of "frames", which are used extensively in expert systems for knowledge representation. If software parts are thought of as objects, a frame-based system can be built which contains declarative and procedural information about parts.

The knowledge contained in such a frame would be symbolically stated, using a formal grammar. The grammar of the frame will state the function of the part, such as number and types of exceptions, real-time requirements, accuracy, etc. If a hierarchical representation is used to describe the system, attributes of parts can be "inherited" from other parts at a higher level in the hierarchy. An expert system can then be built to compare requirements to information about parts, yielding a probabalistic measure of applicability of a part to a problem. The more information available about a part, the better a measure of applicability can be determined.

Another technology that can be applied to reusability is that of Archetyping. (1) Archetype comes from the latin archetypum for "first molded as a pattern; exemplary". In this case, software specialists capture software soon after it has been tested and delivered, work with users of the software system, and sketch out future requirements for systems of this type . Thus, a team of software and domain experts develops a pattern from which future systems can be generated. The result of an archetyped software system is a template that requires a tool to "fill in the blanks" to customize the software for an application. One such tool is the DARTS technology, offered by General Dynamics. Archetyped software overcomes all the limitations found with Ada generics. Archetyped part elements, combined with a formal grammar, will provide context-sensitive expansion of specifications into compilable Ada source code.

It is the conclusion of this paper that absolute certification is a desirable but extremely difficult to achieve goal. Partial certification is a more realistic goal and is attainable with existing technologies.

Moreover, in order to use parts "as is", they must be kept small and uncomplicated, otherwise the process of certification becomes very complex. A methodology, such as archetyping, combined with the proper tools, can make parts adaptable, reduce complexity, and allow for reuse of larger bodies of code.

The concepts described in this paper reflect research being performed at General Dynamics Data Systems Division, San Diego, California.

(1) Przybylinski, S. "Archetyping- A Knowledge-Based Reuse Paradigm" April, 1986

N89-16333

DEVELOPMENT OF AN ADA* PACKAGE LIBRARY

Dr. Bruce Burton and Mr. Michael Broido

Intermetrics, Inc.
Aerospace Systems Group
5312 Bolsa Ave
Huntington Beach, California 92649

ABSTRACT

A usable prototype Ada package library has been developed and is currently being evaluated for use in large software development efforts. The library system is comprised of an Ada-oriented design language used to facilitate the collection of reuse information, a relational data base to store reuse information, a set of reusable Ada components and tools, and a set of guidelines governing the system's use. The prototyping exercise is discussed and the lessons learned are presented. Our experiences in developing the prototype library and lessons learned from it have led to the definition of a comprehensive tool set to facilitate software reuse.

* Ada is a trademark of the U.S. Department of Defense (AJPO).

INTRODUCTION

With the rising demand for cost-effective production of software, software reuse has become increasingly important as a potential solution to low programmer productivity. In the Ada programming language, explicit support is provided for software reuse through the "package" and "generic" language features. Unfortunately, the concept of Ada software reuse is not a panacea for our current software productivity problems. The notion of software reuse has been popular for decades. But implementing high degrees of reuse has usually failed, with the exception of some efforts in fairly narrow areas (business and compiler applications). The challenge then, is to recognize the contributions that the Ada language can make to a software reuse effort while at the same time identifying and resolving language-independent problems. Based on the promise of the Ada programming language we undertook the development of a prototype Ada package library.

The prototyping exercise included:

- o an examination of the reasons for low software reuse in the past,
- o identification of activities and tools which would support a reuse methodology that spans the software development life-cycle from requirements through maintenance,
- o the development of a phased implementation plan for software reuse that defines a development path from prototype to an operational, multi-company, geographically distributed system,
- o development of a prototype for that methodology,
- o the development, acquisition, and evaluation of representative package entries, and
- o an examination of user interface techniques that could be used to maximize communications between a reuse system and its users.

BACKGROUND

As discussed above, software reuse is not a new concept. Significant efforts have been underway since the early 1960's to improve software development productivity through reuse (consider the early observations of McIlroy about the benefits of reuse presented at the NATO Software Engineering meeting in Garmish in 1968) [STANDISH83]. An analysis of the problems attending reuse has led to the identification of several potential hindrances to reuse [STANDISH83, BROIDO85]. These impediments to reuse can be categorized as technical, economic, and political obstructions. Some typical problems that hinder reuse include:

- o lack of universal standards for component composition, level of documentation, coding techniques, testing, etc.,
- o difficulty in transferring an understanding of the purpose of a software routine from the author to the potential reuser,
- o higher initial development costs and longer schedules,
- o risk management issues such as warranty, liability, and accountability,

- o the "not invented here" syndrome, and
- o the lack of pride typically exhibited when reuse has been selected in a software development project over original development.

While the problems impeding reuse are significant, the large size and cost of a major software development effort provides substantial motivation to improve productivity through reuse. Although Ada provides a natural vehicle for encouraging software engineering reuse, the same technical and political obstructions that have limited reuse in the past are likely to once again impede the sharing of software engineering products across projects. The Software Technology Department within Intermetrics is actively investigating the problems that hinder reuse. We are determined to find solutions to these problems and to collect and reuse Ada packages.

APPROACH

Along these lines, we have defined a phased approach to the development of a reusable package library suitable for use on large Ada applications projects. Rather than define an elaborate reuse facility and implement the library in a single step, we are currently prototyping parts of this facility to investigate the potential

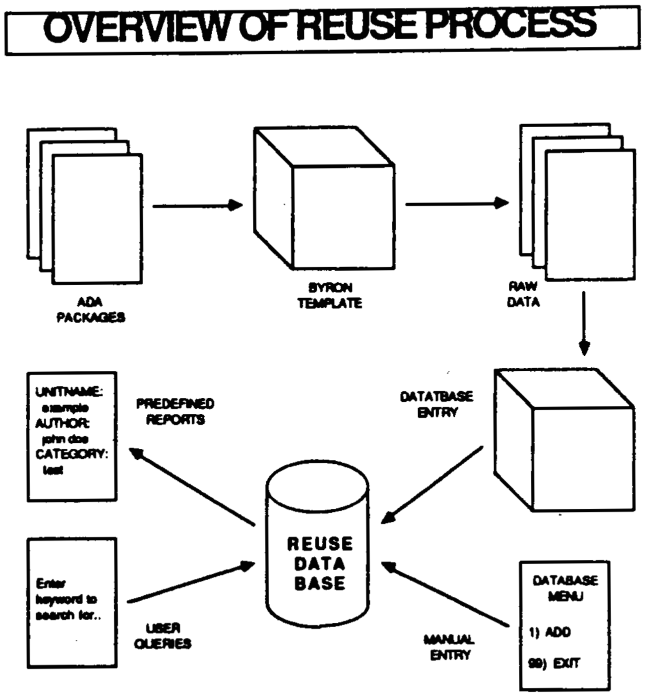


Figure 1. Reuse process overview.

utility of our approach. A complete description of this phased development plan is offered in [BURTON85]. The initial effort on this project has been focused on the creation of an Ada Software CATalog (ASCAT).

An overview of the ASCAT portion of the Ada package reuse system is shown in Figure 1. The system has been implemented using Byron*, Intermetrics' Ada-based program design language, and a commercial relational database management system. Central to the system is the ability of Byron to support definition and use of user-defined keywords.

Software Classification and Data Element Selection

One key to the success of

any reuse scheme is the types of classifications assigned to entries. The primary purpose of these classifications is to facilitate retrieval, but they may also be used to assist in defining storage strategies as well.

Selecting the classifications to be used is really a subset of a larger question: what data elements do we want to be able to retrieve about a particular entry? The list of storable elements seems in our opinion to be highly influenced by the size of the library (number of program units stored) and the degree of cooperation (or potential antagonism) among the users of the library. An initial cut at such a list was prepared [BROIDO85] from the perspective of our ultimate (multiple sites, multiple organizations, multiple usage types) system. Over 60 items which could potentially affect the suitability of an entry were named in seven major categories: identification (3 items), description (16 items), component parts (20), environment/usage (9), ordering information (7), and revision history (11). Even at this length, we recognize that there are undoubtedly many other items which could be added.

This list was far too large for our prototype, so we examined the context in which the prototype would operate. We characterized our initial environment as follows:

- All the users would be from the same company, although there would be several divisions using the common library. Thus, no restrictions on access would need to be supported.

- All initial entries would be written (when possible) in machine-independent Ada, so the compilation and execution environments would be well-defined.

- Source code would always be available, so users could do their own tailoring (no "black boxes"). Support in the form of corrections and training (other than by reading the source code) would not be provided.

- Emphasis was centered around the collection of reusable Ada packages rather than complete programs. Two factors influenced this decision. The first is that most of the packages we wanted to include already existed prior to the start of our efforts, and coherent design documents were not always available. The second factor was the widely distinct set of users we were addressing; they do not share the commonality of purpose which makes

* Byron is a trademark of Intermetrics, Inc.

domain analysis an effective top-down approach. The decision to center our design on packages enabled us to define a standard header for each package, based on the requirements of our Byron program product. Formalized requirements and design documentation were not required.

This decision causes the library to be more supportive of "bottom up" software construction techniques than most of today's top-down methods. The top-down methods reflect an attitude of defining what would be a perfect system and do not adequately recognize the influence of existing tools (including code) should have on requirements formulation in the presence of real cost constraints. (Note that the "object oriented design" strategies that are emerging with Ada reflect a tendency away from strict top-down methods.)

- No a priori naming conventions were established, although an informal guideline was prompted by the technical monitor of one of the contributing programs.

- Configuration management was not rigidly enforced, except within the rules imposed by Ada. In particular, no computerized list of outstanding users (people or programs) of the library routines was maintained.

- The programs which were intending to take advantage of the library provided no <u>explicit</u> funding for tool support or to ensure that any new packages created were generalized and otherwise suitable for future reuse. Package headers and other programmer-supplied information had to be easy (in both time and difficulty) for the programmers to supply.

- Various standards were established for the data items we would collect. Since we were attempting to catalog packages which had been previously created to support several different projects, It was necessary to retrofit many of the selected packages to include the required Byron comments. Part of our evaluation will be to try to identify the difficulties caused by "loose" definitions of essentially narrative fields (e.g., overviews). In addition, no common methodology had been established, so the degree of formality and the list of available support items (repeatable test cases, previous sample output, user documentation, etc.) also varied considerably.

We filtered the original list down to the following data items for the database (others, such as the calling conventions and parameters, would be available from the source code if not given in the overview):

1. Unit name
2. Author
3. Unit size
4. Source language
5. Date created
6. Date last updated
7. Category code (see below)
8. Overview
9. Algorithm description
10. Errors/exceptions generated
11. Up to 5 keywords (for retrieval)
12. Machine dependencies (if any)
13. Program dependencies (if any)
14. Notes

Our retrieval strategy was based upon a combination of two alternate mechanisms. The first mechanism was the assignment of a hierarchical category code, with the hierarchy defined ahead of time and changeable only at well separated time intervals. This scheme is similar in concept to the ones used by Computing Reviews [ACM85] and the IMSL library [IMSL76]. But it was necessary to invent our own classification scheme since neither of those two was suitable to our purposes. Our scheme has the advantage that everyone knows what the codes are and can use an effectively finite procedure for searching the entries. Disadvantages include a growing list of vastly dissimilar "miscellaneous" entries and the inability of the original hierarchy designers to provide sufficiently discriminatory categories to provide effective retrieval (not too many or too few candidates).

For the second mechanism, we allowed the submitters to supply up to five keywords to be associated with each package. These keywords are not associated (as implicitly occurs within the hierarchy of categories), allow for overlapping topics (the packages do not conveniently fall into strict tree classifications), and can grow (without reprogramming or an all-knowing database administrator) with the needs of the projects they are created for. A scheme similar to this has been employed on NASA's COSMIC (Computer Software Management Information Center) system on complete programs, although the keywords allowed are suggested by the program authors and filtered by an acceptance team.

One of the authors is a member of the Applications Panel of the Department of Defense's Software Technology for Adaptable, Reliable Systems (STARS) Program. An important open issue surrounding the formation of a potential Ada package library to be available as GFE materials for DoD contracts is defining the quality of the entries. On the one hand, some people advocate including only items of the highest quality, with full DoD standard documentation and even formal independent validation and verification (IV&V) required on new entries. Others prefer to let a more flexible scheme apply, with a "trust level" associated with entries. This latter scheme encourages "promotion" of existing entries from "buyer beware" to higher trust levels; after all, using informally qualified designs and code and then adding formal testing and documentation can still take less time (and often risk) than inventing from scratch. For the prototype, we decided to let all submitted entries be accepted and then evaluate the impact of this decision.

Reuse Information Extraction Mechanism

Another critical phase in the development of an Ada package library involves the extraction mechanism used to collect reuse-oriented information. The extraction mechanism utilized in an Ada package library must eventually provide several different capabilities to insure efficient operation. These required capabilities include:

- o support for automatic data collection,
- o support for insuring standardization of data entries,
- o support for assuring continuity and consistency of reuse information across the Software Development Life Cycle (SDLC),
- o support for checking completeness and reasonableness (e.g., dates), and
- o support for reuse information examination.

The reuse information extraction approach utilized in our Ada package library is detailed in Figure 2. An analysis of this figure reveals that each of the elements previously identified for data collection has been mapped into predefined or user-defined keywords for the Byron design tool. A Byron template program was subsequently developed to automatically extract the reuse-oriented information. This information is placed into a file that can be directly processed into the ASCAT data base.

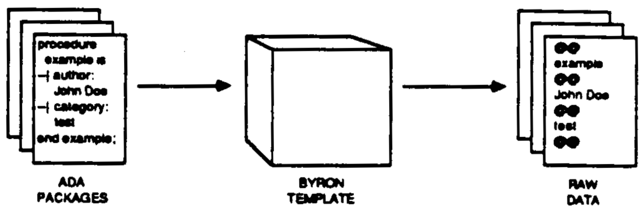


Figure 2. Extraction mechanism overview

The use of a Byron-oriented reuse information extraction mechanism provides most of the required capabilities enumerated above. This approach provides a means for automatic collection of data standardized in field name and format. Since the Byron design file is intended to transition into the implementation with reuse data intact, support is offered to assure information continuity across multiple phases of the SDLC.

While this extraction approach has many positive features, it is not without its shortcomings. The lack of predefined reuse attributes within Byron fails to support direct examination of reuse data items for completeness, consistency, and reasonableness. The inclusion of reuse-oriented information into the Byron-produced program library represents a simple potential improvement to our approach that could aid in the examination of the reuse data items.

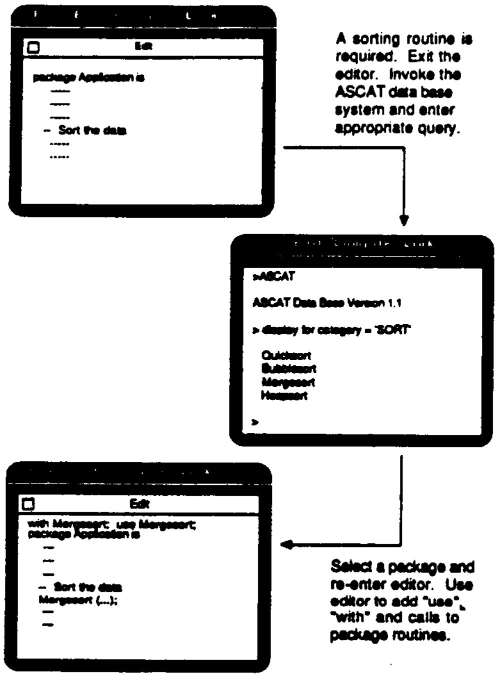
Software Catalog Implementation

The software catalog for the reusable package library was implemented through the use of a commercial relational data base management package. The data definition capability used for field definition and the built-in data base programming language facilitated the examination of reuse data for limited correctness and consistency checking. The use of a data base also aided in the rapid development of an interface between the software catalog and potential Ada package users through the utilization of predefined reports and support for ad hoc user queries. Nonetheless, the user interface represents a weak link in our prototype package library. The present interface is very limited in the sense that it offers no context-specific support for communication between the reuse system and its users.

The present software catalog is limited in its interaction with the user. For example, consider the scenario of a software engineer performing an application software design of a routine that requires a sorting package. In the present system, the software engineer would need to: 1) exit the editor, 2) enter the software catalog data base system, 3) enter a query to identify the available sorting packages, 4) select the desired package, and 5) re-enter the editor and issue the necessary commands to draw the desired package (design/-code) into the applications program design.

This initial prototype software catalog can readily be improved to enhance the way in which it interacts with user. In Figure 3, the present mode of interaction is depicted. In Figure 4, another potential scenario is shown. In this scenario, a multi-window environment is used where the user may perform the software catalog inquiry and concurrently examine several promising packages without exiting the editor.

A third possible operational scenario of the software catalog is not pictured. In this third approach, the data base query language would be replaced by a natural language front-end, the software catalog search would be assisted by an expert system, and the multi-window


Edit
A sorting routine is required. Exit the editor. Invoke the ASCAT data base system and enter appropriate query.
Select a package and re-enter editor. Use editor to add "use", "with" and calls to package routines.
Edit

```
package Application is
   -- Sort the data
```

```
>ASCAT

ASCAT Data Base Version 1.1

> display for category = 'SORT'

  Quicksort
  Bubblesort
  Mergesort
  Heapsort

>
```

```
with Mergesort; use Mergesort;
package Application is
   -- Sort the data
   Mergesort (...);
```

Figure 3. Current ASCAT operational scenario

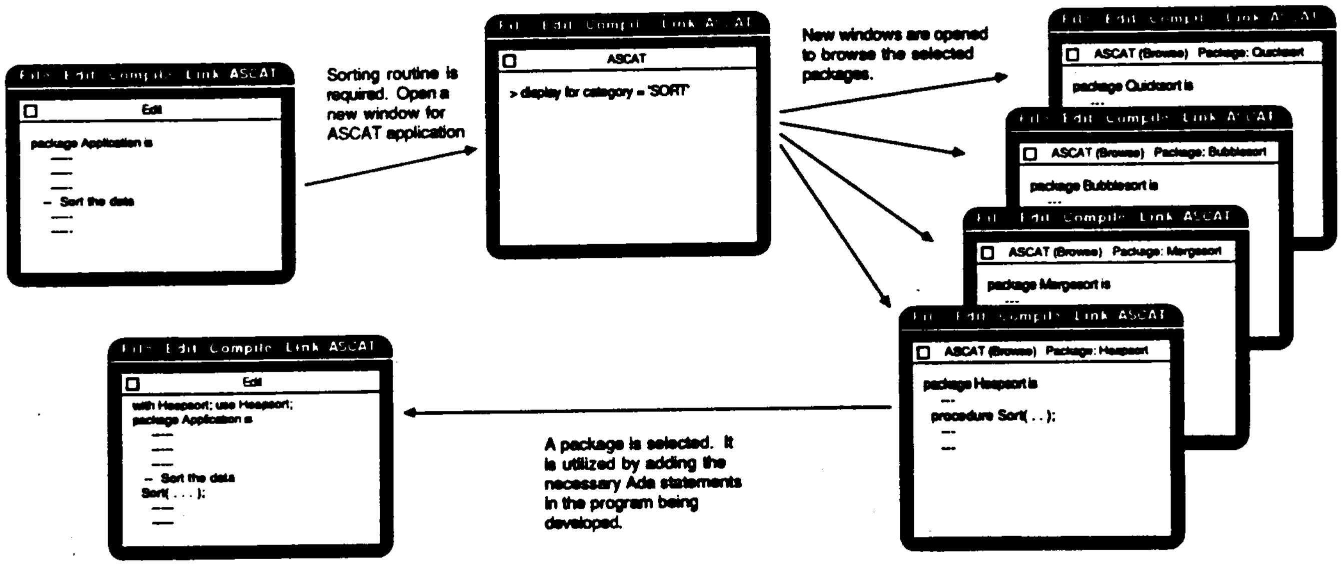
approach would be supported by a language- and context-sensitive editor. The third approach is feasible with investigation into its implementation occurring in several current projects [ANDERSON85].

Intermetrics is currently investigating the implementation of this third approach. We are integrating a commercial natural language language front-end on our reuse database and are designing an expert system to facilitate evaluation and selection of alternative Ada packages. Although it is premature for significant conclusions on our expert system efforts, we have made several observations about the advantages and disadvantages of the Natural Language Front-End (NLFE).

Our preliminary findings on the NLFE are not surprising. As expected, we found the NLFE to be significantly easier to use than the traditional database query language supplied with our DBMS. On the negative side, we found that the natural language interface was substantially slower than our traditional database query language. Our initial figures show a performance penalty associated with the NLFE which ranged from a factor of five for relatively simple queries to a factor of ten for fairly complex queries.

Our preliminary query composition comparisons and initial performance evaluations show that the NLFE approach is a viable alternative to traditional database query languages. We are currently addressing the performance issues that plague the NLFE

ORIGINAL PAGE IS
OF POOR QUALITY


File Edit Compile Link ASCAT
Edit
ASCAT
ASCAT (Browse) Package: Quicksort
ASCAT (Browse) Package: Bubblesort
ASCAT (Browse) Package: Mergesort
ASCAT (Browse) Package: Heapsort

```
package Application is
   --
   --
   --
 -- Sort the data
   --
   --
```

Sorting routine is required. Open a new window for ASCAT application

```
> display for category = 'SORT'
```

New windows are opened to browse the selected packages.

```
package Quicksort is
   ---
```

```
package Bubblesort is
   ---
```

```
package Mergesort is
   ---
```

```
package Heapsort is
   ---
 procedure Sort( . . );
   ---
   ---
```

A package is selected. It is utilized by adding the necessary Ada statements in the program being developed.

```
with Heapsort; use Heapsort;
package Application is
   --
   --
   --
 -- Sort the data
 Sort( . . . );
   --
   --
```

Figure 4. Improved ASCAT operational scenario

approach. We feel that the application of NLFE and expert system technology to the software library area will significantly simplify the operation of a software library and substantially improve the productivity of the software library users.

LESSONS LEARNED

The development, collection, evaluation, and cataloging of reusable components and tools undertaken in the development of an Ada package library has led to some interesting observations concerning Ada package reuse. Unfortunately, we do not yet have enough experience to evaluate the selected category scheme, keyword retrieval capability, or the list of collected data elements.

During the past year, we have developed a set of test and analysis tools written in Ada and intended for Ada software development efforts. The fixed-price nature of this contract and the fact that it represented the first major Ada development contract within our division motivated us to emphasize reuse of existing Ada packages as a cost and risk reduction measure. Based on the results of that contract, we found that reuse of existing generic support packages significantly improved our productivity, with over 33% of the code comprised of reused packages.

On the negative side, we found that several of the tools initially exhibited poor performance. In almost every instance, we found the general nature of the reused packages to contribute heavily to the performance problems. We also found that the generic Ada packages offered much more functionality than required in our application. The extra functionality resulted in a size penalty with respect to the executable code. The use of a performance analyzer and tailoring of the reused code for the current application substantially improved tool performance [RATHGEBER86].

We also studied the problem of composing reusable applications packages from existing reusable components. As part of an Air Force study, we compared the performance of two different implementations of reusable Kalman filter routines. One of the routines was written in Ada; generic Ada mathematics packages were heavily used in its development. The other routine was written in FORTRAN and specifically designed to solve a specific Kalman filter problem. A performance comparison of the generalized Ada package against the custom-tailored FORTRAN routines showed the FORTRAN routine to exhibit significant speed advantages over its Ada counterpart. This performance difference is probably due to the relative immaturity of the Ada compiler used in this study and also to the generalized nature of the Ada packages. An important conclusion of the study is that the performance problems associated with including a generalized reusable Ada package into an applications program are substantially compounded when an entire system is comprised of reusable components which also consist of reusable components.

Although many of our lessons learned have negative implications for the use of Ada reusable packages, there is some light at the end of the tunnel. Reuse was a big aid in increasing our productivity in the development of Ada test and analysis tools. We also found that reuse can be successfully employed in the development of efficient Ada systems if sufficient thought is put into how the packages are to be reused and if the proper tools are available (e.g., such as a performance analyzer).

CONCLUSIONS

In accordance with our previous plan, we have completed a prototype mechanism for extracting reuse information from packages developed in the normal course of business. We also have a primitive mechanism for entering that data in a catalog and searching the catalog for entries that are potentially useful on new projects. The approach centers on the design and implementation phases, since these are the ones to which reuse concepts may most readily be applied in the given environments.

We have confirmed with actual experience our earlier assessment that successful implementation of a reuse methodology requires thought, action and management direction and support throughout the software life cycle. This, however, may require a management reorientation to the view of software development as the acquisition of a long-lived corporate asset rather than as only the work required to produce the current deliverable [WEGNER84, YEH85]. Complementing the reuse efforts being conducted by the STARS office, which are targeted at long range objectives, our approach provides useful tools which can be utilized immediately.

We have achieved some success in applying software reuse. Effective use of the packages forced us to define subsets of them which subsequently required performance tuning. This points out the value of developing a comprehensive reuse methodology, with adequate support tools to facilitate the development of efficient systems comprised of reusable components.

The Ada language and the methodologies growing up around it provide a good start toward achieving larger scale reuse than we have achieved in the past. But they are not enough by themselves. Even with Ada, there are still plenty of obstacles to reuse. A management commitment and desire to improve productivity when coupled with a comprehensive reuse methodology and the proper tools offer substantial promise for improvement.

REFERENCES

ACM85	"Introduction to the CR Classification System," Computing Reviews, Vol. 26, No.1. Association for Computing Machinery, January, 1985, pp. 45-57.
ANDERSON85	Anderson, C.M. and McNicholl, D.G., "Reusable Software - A Mission Critical Case Study", AIAA Computer in Aerospace V Conference, October 21-23, Long Beach, California.
BROIDO85	Broido, Michael D., "Software Commonality Study for Space Station Phase B", Intermetrics Report IR-CA-029, Intermetrics, Inc., 29 May 1985.
BURTON85	Burton, B.A. and Broido, M.D., "A Phased Approach to Ada Package Reuse", STARS Workshop on Software Reuse, April 9-12 1985, Naval Research Laboratory, Washington, DC 20375-5000.
IMSL76	Reference manual, The International Mathematical & Statistics Libraries, IMSL, Fall, 1976.
RATHGEBER86	Rathgeber, R.L., "Technical Report on Ada Test and Analysis Tools", Intermetrics, Inc., Huntington Beach, California, In Preparation.
STANDISH83	Standish, T.A., "Software Reuse", presented at the ITT Workshop on Reusability in Programming, Rhode Island, September 7-9, 1983.
WEGNER85	Wegner, Peter, "Capital- Intensive Software Technology," IEEE Software, Vol. 1, No.3, IEEE Computer Society, July, 1984, pp. 7-45.
YEH85	Yeh, Dr. Raymond T., "Japanese and Brazilian Software Technology Initiatives". (Luncheon address). Published by the NSIA Software Committee in the Proceedings of the First DOD/Industry STARS Program Conference, 30 April 1985 - 2 May 1985.

N89-16334

A DESIGN FOR A REUSABLE ADA LIBRARY

John D. Litke
Grumman Data Systems Corporation
1000 Woodbury Road
Woodbury, New York 11797

ABSTRACT

A goal of the Ada language standardization effort is to promote reuse of software, implying the existence of substantial software libraries and the storage/retrieval mechanisms to support them. We propose a searching/cataloguing mechanism that permits full or partial distribution of the database, adapts to a variety of searching mechanisms, permits a changing taxonomy with minimal disruption, and minimizes the requirement for specialized cataloguer/indexer skills. The important observation is that key words serve not only as an indexing mechanism, but also as an identification mechanism, especially via concatenation and as support for a searching mechanism. By deliberately separating these multiple uses, we achieve the modifiability and ease of growth that current libraries require.

Extensive reuse of software is a goal that industry has found difficult to reach. Among the many issues to be solved before extensive reuse is a reality is the design and construction of a software part storage facility. The requirements for such a system exceed those normally found in a conventional software management system and thus demand a new design approach. This paper proposes a new design for a critical component of a software library, the searching and cataloguing mechanism.

All libraries have a common set of functions to perform. Some well known examples are storage of information, accession, discard of materials, and searching and retrieval of information. Most of these functions for a computerized software parts library can use known computer science or library science methods. However, the function of searching and retrieval has proven difficult and it is in this area that we propose a new design.

As in any library, the function of a searching mechanism is to retrieve any information relevant to a query, not just a precisely specified fact. This means that context and meaning are important elements in the query interpretation process. Furthermore, the information in a library is not static, but rapidly changing both in scope and in terminology used to describe the relevant topics.

A useful searching/retrieval mechanism for a library of software would support inquiry from at least three different points of view. First, it should support the more conventional inquiry by keyword matching to select items by language, machine, author, etc. Second, it should allow searching by conventional topical descriptions in a variety of areas, such as aeronautics, electrical engineering, etc. Third, it should allow searching by algorithmic content, rather than by intended function. For example, we should be able to find an algorithm both by the topical "edge enhancement techniques", and the algorithmic, "fast fourier transforms." If there were a universal topical and algorithmic specification nomenclature, we would need an elaborate, but conventional index. However, all engineering fields have different topical indexing styles that do not map one to one to each other. Further, their algorithmic nomenclature is also not congruent to the nomenclature of other fields. Hence, a complete indexing/classification scheme must contain synonyms, see alsos, analogous references, etc.

If the resulting classification system were stable, the problem is complex enough. Furthermore, not only is the conventional taxonomy unclear and dependent on the engineering field of the user, but also the information that we are indexing is unformatted with an uncontrolled vocabulary.

A superficial solution to this searching and retrieval problem is to propose a computer based database system, complete with proven query languages and report writers.

However, the problem is not one of retrieval of information by unique key word, but retrieval of relevant information given a set of partially applicable key words or phrases. The differences are so substantial that a distinct field of information retrieval has evolved to address the issues as distinct from database technology. A review of the current problems in the field will illuminate the nature of the problem and the import of our proposed solution.

Current research in this field of information retrieval can be summarized in four categories (BART85):

Automatic Indexing
Information Structures
Query Formulation
Query Evaluation.

The workers in the area of automatic indexing are trying to find a way to convert unformatted information with an unstable vocabulary into a formatted, stable vocabulary that can be served with conventional indexing systems of the database community (ABBE75). The heart of the problem is to determine the content of a document by analysis of its text. This problem is analogous to the problems faced by the automatic language translation efforts and has proven very difficult.

The information structures area is trying to determine ways to represent the information contained in documents and the ways that documents relate to one another. There are three principal approaches. The first seeks to develop useful and comprehensive classification mechanisms to apply to all documents. This approach is the established, classic approach of libraries. The second seeks to develop thesauri that capture not only meaning, but relationships in a limited vocabulary of terms. The principal difference of a thesaurus and a classification system is the inclusion of relationships into the scheme, such as "part of", or the narrower/broader relationship of classes, and other ordering relations. Finally, a system that emphasizes relationships will take on the appearance of a semantic net or other network structures. At this end of the spectrum, more meaning is in the relationship between terms, and so the problem of synonyms and partially related terms becomes important.

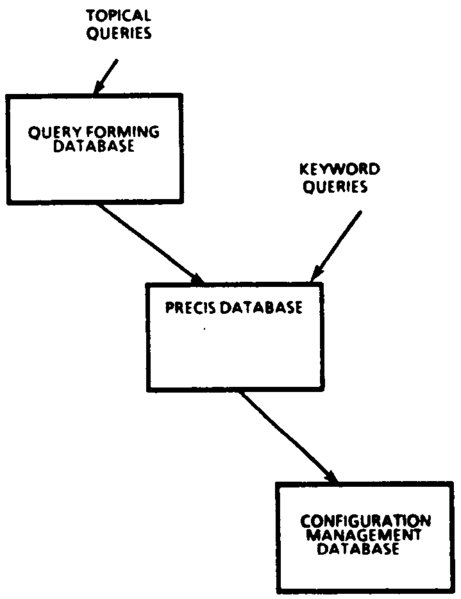
The query formulation area is not independent of the query evaluation area, although it does contain an important component of human interface in addition to the selection logic mechanism. To date there has not been a great deal of success with either boolean logic based systems that allow the formation of "this and that but not the other thing" type of queries, or with combinatoric systems that allow "similar to this or that" type of queries. There are two major difficulties with the boolean logic approach. First, the inquirer must know the allowable vocabulary and indexing scheme so that the query will be well formed. For large, complex systems like a library, this is unlikely for a casual inquirer. Second, boolean logic makes it difficult to specify major/minor selection criteria (BART85), (CROF81), (BOOK85). The combinatoric approach cannot represent boolean constraints easily, often leading to poor selectivity, and difficulty indexing uncertain information.

PROPOSED SOLUTION

The software library searching and retrieval design has several key constraints. First, the automatic classification of entries is beyond a commercially viable solution at present. Since the library will become large, we must minimize the classification effort required until automated solutions are available. Second, the information structures that are desirable are not yet known. Because a library is intended to have a long life, we must minimize the impact of modifications to the information structure or the classification/retrieval structure. Further, it is desirable to distribute widely the classification/retrieval mechanism to the user community to encourage a uniform approach to nomenclature/classification. To make such a distribution practical, the classification mechanism must be separable from the actual documents that it classifies. Finally, a mechanism must be found to classify the subject documents without extensive services from a professional indexer. Since automatic classification systems are still experimental, this implies a reliance on the help of the submitter/author, a heretofore not totally satisfactory source of indexing information. The solution we propose is not a final solution, but rather an architecture that will serve the present and grow gracefully into the future.

Our solution to this problem uses three key ideas supported by three interrelated databases. First, a precis is required from all submitters to minimize the indexing work of the library staff. Second, the classifications required of the author by the precis are guided by a database system supplied by the library to ensure that classifications are reasonable and correct. Furthermore, classification cross references and see alsos are maintained by the library staff, and are not expected of the author. Third, the searching mechanism is separated from the storage/retrieval mechanism to make future modifications to either easier.

The largest of the three interrelated databases is a conventional configuration management system and holds the full text files of all information in the library. (See Figure 1.) Other than indices required by the configuration management function, items are identified by a unique accession number.



Architecture of the Library System

Figure 1.

The second database holds a precis of each logically unique cluster of documents in the library. This precis requires certain information such as author, language, etc. It serves as a brief reference document to a collection of related documents that are logically indexed as one. Such a collection might include the specification, functional design, code, test driver, and test data for a particular module. The precis represents the finest level of granularity that a user can see from the indexing system. Note that the configuration management system is free to control the configuration of such a collection either as a unit, or as separate documents. The design of a precis itself is discussed separately below. The second database is organized conventionally in either a hierarchical or relational model, supporting conventional keyword query mechanisms.

The third database contains the indexing/classification system and serves as a query front end to the precis database. This database supports complex query forms on the classification system, not on the documents that are classified, and enables the user to easily construct complex queries for eventual submission to the precis database. This separation of the querying process from the underlying data containing system will allow change in either mechanism without rewriting the other. It allows users that are satisfied with conventional keyword access to use a conventional system, while allowing independent development of more elaborate searching/retrieval mechanisms.

There are three important properties of this solution. First, separation of the databases allows smaller, indexing databases to be distributed widely without the high cost of distributing large quantities of data in the configuration management database. Second, the version control and dependency relationships are removed from the searching/indexing mechanism to simplify the design and maintenance of the databases. Third, the topical relationships are separated from the precis information. This topical information is stored in a separate, logical tree organization so that variable depth indexing and class/subclass relationships can be maintained with a fixed record and fixed key size indexing scheme. (This was the same problem that led the GRIPHOS database designers to separate the indexing and retrieval mechanism in a similar way.)

This design rests on a plausible, but untested assumptions that the data to be classified is regular enough that:

1. Typical relationships between indexable topical terms are sufficiently generic that the relationships and the data are separable without introducing excessive retrieval error.

2. Classification can be done sufficiently well that the number of false identifications is tolerable. This implies that classification with a fixed vocabulary is possible.

3. The subject matter is sufficiently regular that classifications can be usually identified with a pre-existing taxonomy and that separable relationships are sufficiently generic that significant classes of objects are formed.

A small database is being constructed to test the validity of these assumptions.

PRECIS DATABASE

One of the difficulties with conventional keyword approaches to database access is that the keyword vocabulary must be controlled to eliminate misspellings, synonyms, etc. Further, the indexing terms are not readily expanded or changed without structural database changes. For these reasons, the precis database will be indexed with commonly agreed keys that have a finite, known range and represent quantifiable characteristics. For example, we will index on module name, version, author, machine, language, operating system, etc. This information will be obtained from each submitter to the database by requiring the submission of summary information according to a specified style. From this summary information, a trained indexer will prepare a precis with a controlled vocabulary for inclusion in the precis database, and the original summary information will be retained in the configuration managed database.

To form a connection between the precis database and the query forming database, the author must supply up to five classification codes. These codes will be selected by using the query forming database itself to explore the classification space and will be critiqued by a professional indexer for suitability. The classification codes need not be at a uniform depth of classification.

QUERY FORMING DATABASE

The query forming database bears the responsibility to support sophisticated inquiry into the precis database. There are two problems to be solved, while honoring several constraints. First, we must devise an indexing and classification system that will allow flexible searching of the precis database. Second, we must design an effective human interface to the query forming mechanism that does not require extensive knowledge of the underlying classification system.

To be an effective tool, the solution must also honor several constraints. First, it must support multiple views of algorithm classification for a variety of engineering disciplines. (The notions of subset/superset classifications are especially varied among disciplines.) Second, it must allow retrieval at uniform levels of detail so that broader scope documents are not retrieved together with narrower scope documents unintentionally. Third, it must grow gracefully with minimal or no re-classification of existing software. Fourth, it must be easy to provide classification guidance to submitters of software so that they can provide effective aid to the professional indexers.

The solution that we propose establishes a relationship among classification systems in the query forming database and not among items in the precis database. Thus relationships among library modules are represented implicitly via the classification system and not via explicit links in the precis or configuration database. We require the explicit relationships between elements to be maintained by the management database, since such are more static, while the more dynamic classification relationships are maintained in the classification system itself.

To obtain a classification that is familiar and useful to a wide variety of engineers, we propose a hierarchy of overlapping classification systems, whose inter-relationships are determined and maintained by expert indexers. The top level classification system will use the taxonomy of Dissertation Abstracts for field identification. Under each field, the indexing scheme customary to that field will be used. For example, mechanical engineers would use the scheme of the Applied Mechanics Reviews, Computer Scientists would use Computing Reviews, etc. The depth of the indices will vary from field to field. Since some areas of knowledge such as algorithms are found very frequently in software systems, a finer classification resolution such as that provided by the CALGO system will be needed. It will be up to the professional indexing staff to provide cross references and see also relationships among and between the classifications. The only change that will be made to the adopted classification system is to apply a more uniform numbering system to adjacent levels. This allows the database retrieval mechanism to refer to any classification with a unique code.

This design allows multiple classification schemes to exist and be interrelated with no overt cooperation from authors. The scheme can be revised and extended with little or no alteration of existing software. Deleted classification terms are simply translated by automated means into designated alternate categories, while new terms require reclassification only incrementally. (The simultaneous deletion of an old classification and re-assignment to a variety of new ones would require some re-classification.)

The system allows a user to self classify a module in his/her own vocabulary, while automatically supplying the cross references to other vocabularies and fields. Since the classification database is physically separate from the precis database, users can choose to use printed versions of the classification schemes and more laboriously search the precis database by conventional keyword selections. Finally, since a mechanism is provided to search the classification system itself, the classification can be made much more elaborate and precise without making the searching job harder.

Since the whole intent of the query forming database is to enable the user to more readily search the library, the quality of the user interface will be highly dependent on good physical implementation as well as a good match to people's searching strategies. We envision that the user will be able to select categories and sub-categories as she navigates freely through the classification system, able at any time to retrieve some or all of the precis implicitly selected, refine the query by elaboration, deletion, or addition of additional keyword specific qualifiers (such as language) as required. A modern, menu driven system with mouse and optional text entry is a candidate interface paradigm.

Each entry in the classification database will contain a count of the number of entries in the precis database that are indexed by that classification. This will allow a user to sharpen the query procedure without necessarily accessing the large precis database. Items can also be classified to any depth that the author deems appropriate. Exhaustive precision in classification is not required.

REFERENCES

(BART85) Bartschi, Martin "An Overview of Information Retrieval Subjects," Computer, (May 1985) pp 67-84.

(CROF81) Croft, W.B. "Document Representation in Probablistic Models of Information Retrieval," J. ASIS, V 32, 1981, pp 451-457

(ABBE75) Abbey, Scott, "GRIPHOS as a Relational Database System," Dissertation 1975, State University of Stony Brook, Stony Brook, New York.

(BOOK85) Bookstein, Abraham "Implications of Boolean Structure for Probabilistic Retrieval" Proc 8th International ACM SIGIR Conf, Montreal, Canada, June 5-7, 1985 pp 1177.

Designing Generics for Compatibility and Reusability

Introduction

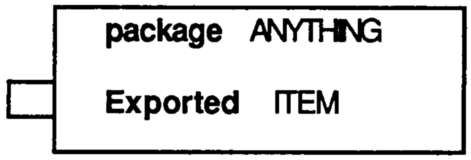
Many desirable features may be achieved by implementing a consistent design in generic libraries. The techniques discussed here are not exhaustive, but can form the basis for a design. In addition, a few of the simpler uses of generics will be touched on.

One of the major goals of generics is reusability. It is toward this end that most of these techniques are directed. Reusability is desirable not only across applications, but also across data structures; i.e. a process applied to a linked list should be available for an array also. With this in mind, most of the applications of the generic library should treat the data structure as a single entity and provide routines to apply to that data structure.

It is difficult to understand generics without using them. Consequently, these discussions are directed to those who have some practical experience with generics. In addition, it is recommended that the reader take the time to try his hand at some of the more difficult examples continuing to look for more ways to turn the specific into the generic.

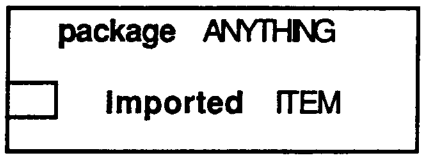
Exporting

Exporting refers generally to any visible component of a package. This includes types, objects, procedures, functions, (sub)packages, and generics. Specifically for generics, any item that becomes visible at instantiation is considered exported. Along the lines of the notation suggested by R.J.A. Buhr, this can be represented graphically [BUHR],



Importing

Importing most often refers to the use of items from another package made visible through a **with** clause. However, this paper will refer to parameters required for instantiation of the generic as imported during instantiation. Importing can be similarly represented graphically.

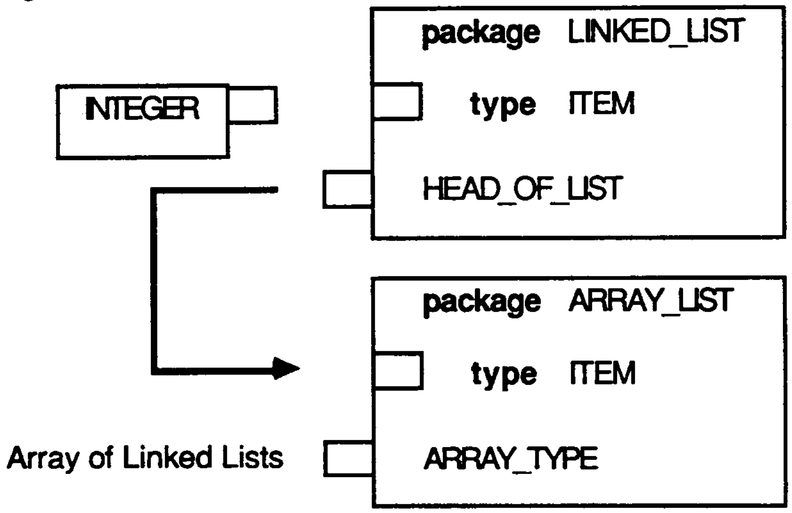


Notation

Each box represents not only what is imported and exported, but also the order in which the individual structures are available. In the case of the generic, all items must be available for importing before they can be exported. In addition the name of the module appears at the top.

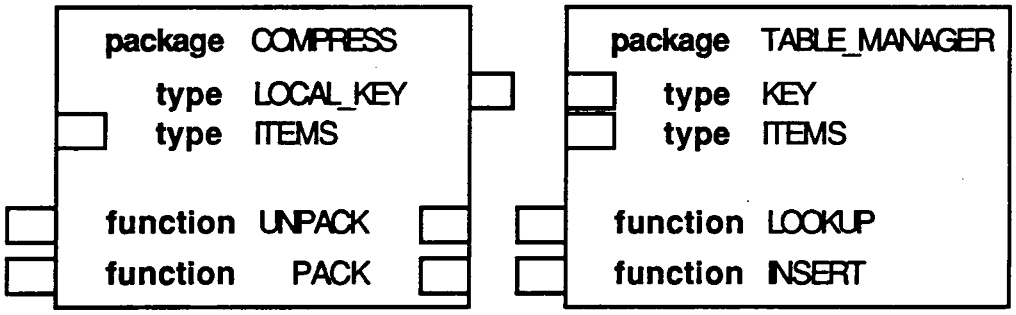
Availability

A feature that may be overlooked by a first glance at generics is that the functions and types visible in a generic are immediately available after an instantiation to be used in further instantiations. Assuming the existence of a linked list package and an array manipulation package, an array of linked lists can be created:



Embedding

Another feature is to instantiate a generic within a generic. Although a data type is imported to a generic routine, it may be used in another instantiation as well as any functions or procedures to manipulate that type. For example,



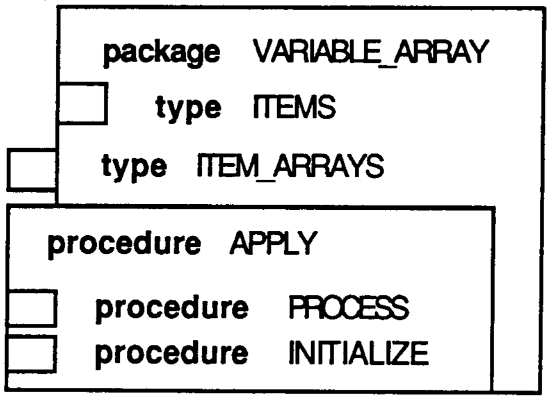
```
with TABLE_MANAGER;
generic
    type ITEMS is private;
package COMPRESS is
    subtype LOCAL_KEY is INTEGER;

    package LOCAL_TABLE is new TABLE_MANAGER( LOCAL_KEY,
                                              ITEMS );
    function UNPACK ( KEY  :  LOCAL_KEY) return ITEMS
        renames LOCAL_TABLE.LOOKUP;
    function PACK    ( ITM  :  ITEMS)      return LOCAL_KEY
        renames LOCAL_TABLE.INSERT;

end COMPRESS;
```

In fact, any imported item may be used in further instantiations. Constants, functions, procedures, etc. can all be used to instantiate an embedded generic.

The only restriction in the Language Reference Manual is the use of recursive instantiations as discussed in section 12.3. However, there is a way of using the results of an instantiation in the instantiation itself. This will be discussed later.

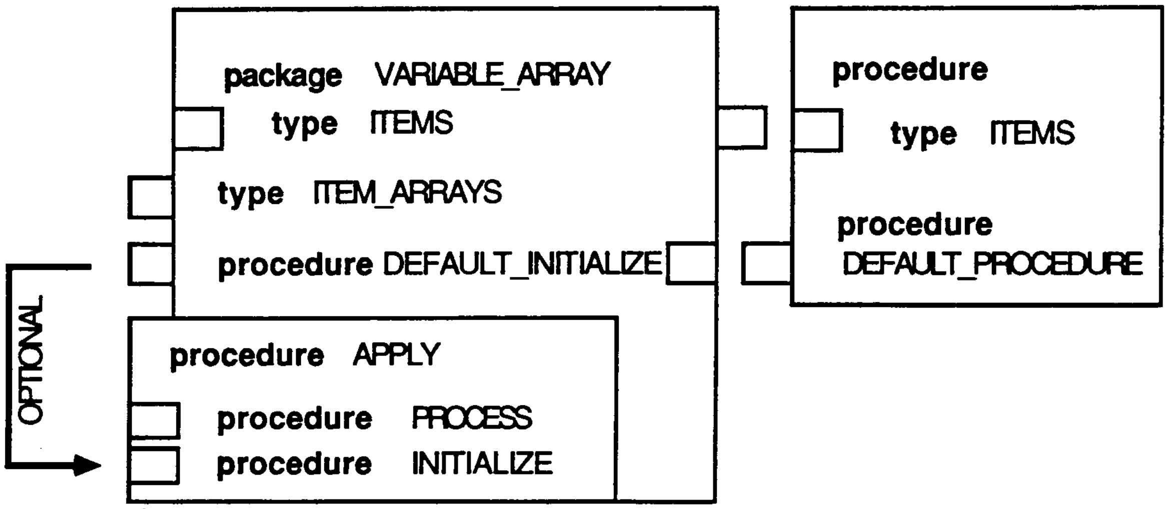
Exporting Generics

Exporting generally refers to any data type, object, function or procedure that is visible in a package. Specifically, generics themselves can be exported from generics. As opposed to embedding, exporting makes a generic visible for further instantiation. The most obvious example is a generic used to operate on the whole data structure which has been given the process to APPLY to each item.



```
generic
    type ITEMS is private;
package VARIABLE_ARRAY is
    type ITEM_ARRAYS is array( POSITIVE range <>) of ITEMS;

    generic
        with procedure PROCESS    ( ITEM : in out ITEMS);
        with procedure INITIALIZE ( ITEM : in out ITEMS);
    procedure APPLY( ITEM_ARRAY : in out ITEM_ARRAYS);
end;
```

For flexibility, the exported generic can easily consist of default parameter values. This precludes the necessity for constantly writing null procedures and functions to match generic arguments. If you like, the default procedures and functions can actually be instantiations themselves.



```
generic
    type ITEMS is private;
procedure DEFAULT_PROCEDURE( ITEM : in out ITEMS);
--begin
-- null;
--end;

with DEFAULT_PROCEDURE;
generic
    type ITEMS is private;
package VARIABLE_ARRAY is
    type ITEM_ARRAYS is array( POSITIVE range <>) of ITEMS;

    procedure DEFAULT_INITIALIZE is new DEFAULT_PROCEDURE( ITEMS);
    --This procedure could be inline

    generic
        with procedure PROCESS    ( ITEM : in out ITEMS);
        with procedure INITIALIZE ( ITEM : in out ITEMS)
            is DEFAULT_INITIALIZE;
    procedure APPLY( ITEM_ARRAY : in out ITEM_ARRAYS);
end;
```

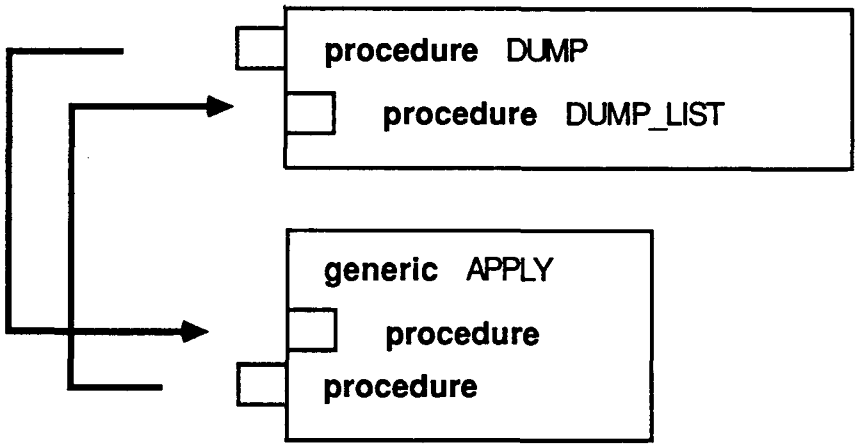
Indirect Recursive Instantiations and Mutual Dependency

One of the big problems in handling data structures is the inability of generics to handle different data structures. Normally, a generic to handle arrays is different from the generic to handle linked lists, even though the two generics may do functionally the same thing. There is a way to remove this structure dependency from the generic.

It is easier to introduce the technique with an illustration. The first example introduces a problem in compilation order.

```
with LINKED_LIST;

type EMPLOYERS;
type EMPLOYERS_PTR is access EMPLOYERS;

package EMPLOYEE_LIST is new LINKED_LIST( EMPLOYERS_PTR);

type EMPLOYERS is
    record
        NAME        : STRING ( 1..32);
        EMPLOYEES   : EMPLOYEE_LIST.NODE;
    end record;


procedure DUMP( EMP : EMPLOYERS_PTR) is
begin
    PUT_LINE   ( EMP.NAME);
    DUMP_LIST( EMP.EMPLOYEES);    --Not a visible routine yet
end;

procedure DUMP_LIST is
    new EMPLOYEE_LIST.APPLY( DUMP);
```

Notice that there is a way around this interdependency by deferring the procedure definition. This will lead us to a solution to the basic problem of manipulating data structures in a generic. The goal is to use a function or procedure from a generic within a function or procedure that is used by the generic. Graphically,

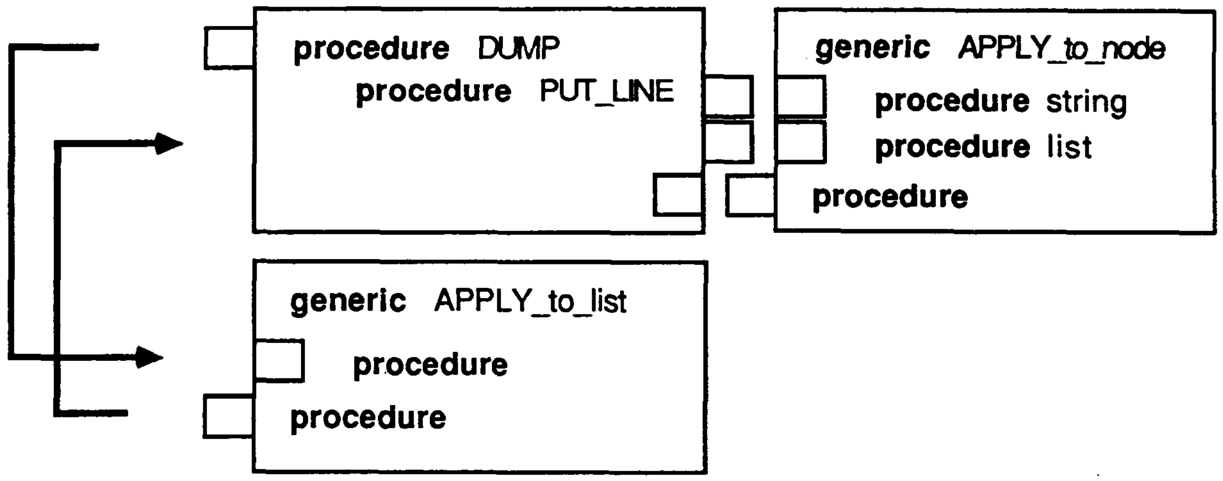


To achieve such interaction, the corresponding *incomplete* declarations of procedures and functions can be used: 1) the actual function is declared, 2) the instantiation of the generic which uses the actual function is done, (making any functions or procedures in the generic available.) 3) Now the actual function can be completed, using any routines available from the generic.

```
procedure DUMP( EMP : EMPLOYERS_PTR);

procedure DUMP_LIST is
    new EMPLOYEE_LIST.APPLY( DUMP);

procedure DUMP( EMP : EMPLOYERS_PTR) is
begin
    PUT_LINE   ( EMP.NAME);
    DUMP_LIST( EMP.EMPLOYEES);
end;
```

At this point, a significant distinction can be made between the normal procedure and the procedure achieved through instantiation-- DEFERRED DEFINITION! Deferred definitions allow normal procedures to interact with each other in ways that instantiated procedures cannot. However, there is a way that this feature can be added to instantiations.

Using essentially the same technique, hide the point of interaction inside a procedure and defer its definition. Remember, although the declaration of an instantiation cannot be deferred, it can be hidden inside an actual procedure which can!

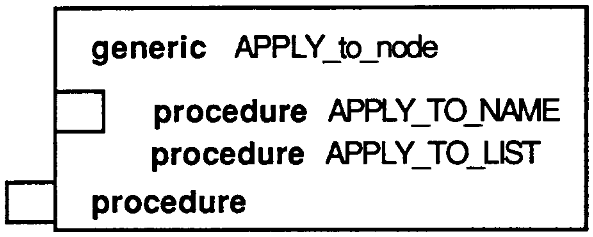


```
    generic
        with procedure APPLY_TO_NAME( STR : STRING);
        with procedure APPLY_TO_EMP   ( LST : EMPLOYEE_LIST.NODE);
    procedure APPLY( EMP : EMPLOYERS_PTR);

procedure DUMP( EMP : EMPLOYERS_PTR);

procedure DUMP_LIST is
    new EMPLOYEE_LIST.APPLY( DUMP);

procedure DUMP( EMP : EMPLOYERS_PTR) is
    procedure INTERNAL_DUMP is new APPLY( PUT_LINE, DUMP_LIST);
begin
    INTERNAL_DUMP( EMP);
end;
```

Data Structure Independence

One of the goals toward compatibility has been reached--removing data structure dependency from the generic. The resulting generic is derived by recognizing the only actual parameter to the application: the procedure applied to the string component of the record structure.

The resulting code is deceptively simple. Almost all of the techniques discussed to this point, and then some, are being used here. In addition, the parameter passed to the generic is actually hidden from the APPLY_TO_LIST procedure. This technique will be discussed more generally later.



```
generic
    with procedure APPLY_TO_NAME( STR : STRING);
procedure APPLY( EMP : EMPLOYERS_PTR);

procedure APPLY( EMP : EMPLOYERS_PTR) is
    procedure APPLY_TO_LIST is
         new EMPLOYEE_LIST.APPLY( APPLY);
begin
    if EMP /= null then
         APPLY_TO_NAME ( EMP.NAME);
         APPLY_TO_LIST  ( EMP.EMPLOYEES);
    end if;
end;
```

Compatibility

In the previous example, the data structure supplied by the generic is now independent of the implementation. If arrays were the hidden implementation inside the linked list package, there would be no code change. But then that is not what is really desired. Preferably if a change is needed, then the instantiation uses a different package. So now the issue of compatibility needs to be addressed.

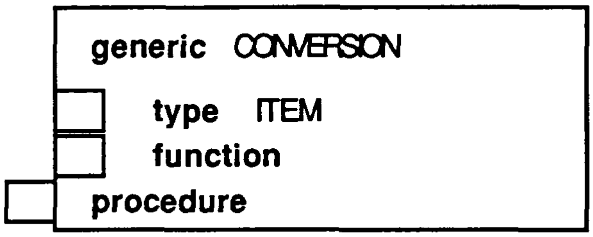
Compatibility for generics can mean different things. The intent of the application must obviously match the functionality of the generic. This may not be enforcable by the language and therefore must be left to design considerations. But for generics that are functionally equivalent, some language features define another meaning for compatibility.

Parameter profile matching and general type matching are required for instantiation. Compatibility occurs when the profiles match as described in the Language Reference Manual 6.6. Keeping a consistent format allows the results from a generic instantiation to be used in another instantiation. It also allows switching between generics with minimal code changes.

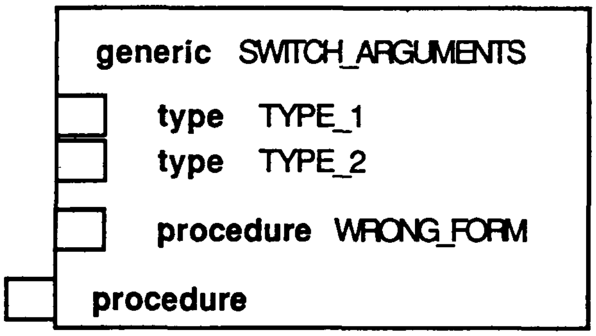
In the cases where generics cannot keep such a consistency, the alternative is to overload the routines from a generic to be both in a procedural form and a functional form. Such overloading then allows the programmer to continue plugging the routines into other generics since Ada will resolve to the compatible form. Additional overloading and default parameters can provide even more versatility.

Profile Conversion

Even when parameters are not compatible, generics can provide a quick solution. A set of profile conversion generics can come in handy:



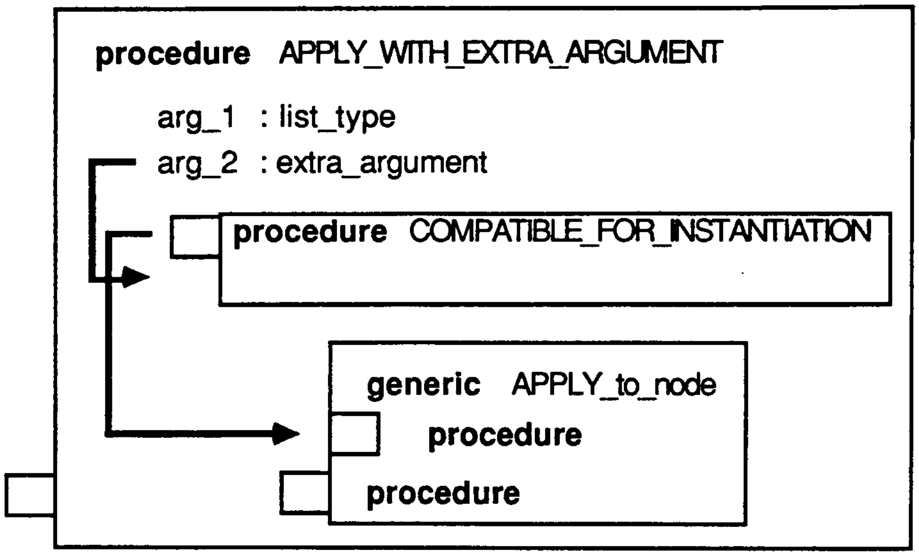
```
generic
    type ITEMS is private;
    with function PROCESS( ITEM : ITEMS) return ITEMS;
procedure PROCEDURAL_PROCESS( ITEM : in out ITEMS);
--  begin
--      ITEM := PROCESS( ITEM);
--  end PROCESS;
```



```
generic
    type TYPE_1  is limited private;
    type TYPE_2  is limited private;
   with procedure WRONG_FORM(  ARG1 : in out TYPE_1;
                               ARG2 : in out TYPE_2);
procedure    SWITCH_ARGUMENTS( ARG1 : in out TYPE_2;
                               ARG2 : in out TYPE_1);
```

Parameter Hiding

Unfortunately, as data structures interact, parameters can get lost in the design. Again, embedding a generic instantiation within a procedure can preserve both the parameter and the needed compatibility.



```
package INTEGER_PACKAGE is new LINKED_LIST( INTEGER);

procedure DUMP( FILE : FILE_TYPE;
                LST  : INTEGER_PACKAGE.LIST_TYPE) is
   procedure INTEGER_DUMP( INT : INTEGER) is
   begin
      PUT( FILE, INT);
   end;

   procedure INTERNAL_DUMP is
      new INTEGER_PACKAGE.APPLY( INTEGER_DUMP);
begin
   INTERNAL_DUMP( LST);
end;
```

Naming Conventions

Since one of the goals of compatible design is to have interchangeable modules, an otherwise unimportant consideration becomes very important. If the functionally equivalent items in one package do not have the same name, then extensive code modifiction will be required to do the conversion. Renaming is an alternative, but obviously a clumsy one.

An example might be a length function provided by a LINKED_LIST package and a TABLE_MANAGER. By using the name LENGTH in both packages, they can be interchanged without modification to every occurrence of the function.

By designing the generics with a consideration toward consistent naming and parameter profiles, switching the implementation becomes trivial--instantiate with the other package. Since the package name does not change, even full dot notation(which is preferable) can remain unchanged. Change the instantiation in the previous example and notice that nothing should change **If** consistent naming conventions are used.

```
package INTEGER_PACKAGE is new VARIABLE_ARRAY( INTEGER);
```

Off The Shelf

The desire to change processing for different data structures and designs indicates that families of generics may be desired to handle the various intentions of the software designer; recursive lists, arrays, ordered lists, ordered arrays, etc.

In fact we can look at it as "plug compatible" software; i.e. the parameter profiles from one generic instantiation match the arguments to another generic. There is still a need for careful design in terms of the application desired. The plugs may fit, but the wrong process may occur.

Conclusion

In all of the discussions so far, the application and the implementation have been general. The techniques apply across a large range of generics such as linked lists, variable arrays, circular lists, ordered lists, etc.

We can now partition the generics into four groups:

1) Generics that create and handle a data structure and in turn provide generic APPLY procedures for treating the data structure as a logical entity.
2) Generics that perform a particular process on any data structure (such as SEQUENTIAL_IO).
3) Generics to manipulate a data structure, processing the data structure for an application that must exploit the internal structure of the data, such as using employee_list.apply in handling the hierarchical nature of the employer data.
4) Generics to handle profile conversion and parameter hiding for compatibility if needed.

Through the use of generics, designed for compatibility and partitioned correctly, a few instantiations can create the application desired. With consistent naming conventions and compatibility, an application can switch from one implementation to another with minimum impact.

Two of the major benefits to a library designed to use these features are maintainability and reliability. Maintainability is enhanced by simply reducing the amount of code to maintain and allowing for interchangable modules. Reliability is enhanced by not only reusing previously tested code, but actually testing the code across many different applications and in different environments.

With these techniques and concepts in mind, a set of generic libraries can more easily be meshed into a compact, compatible unit. Generics can then apply across a greater range of software solutions and integrated into a design effort.

Bibliography

[BUHR] Buhr, R.J.A., "System Design with Ada," Prentice-Hall, Inc., 1984.

Mr. Smith received his M.A. in Mathematics from Vanderbilt University. He has been designing and programming in the Ada language for the last 2 years and has worked extensively with the Generic feature during the development of a successful large scale software project. Mr. Smith is currently working on static analysis techniques in a language independent environment.

N89-16335

Considerations for the Design of Ada* Reusable Packages

by

Norman S. Nise
Chuck Giffin

Rockwell International
Downey, California
June 4, 1986

Abstract
This paper discusses two important considerations that precede the design of Ada reusable packages - commonality and programming standards. First, the importance of designing packages to yield widespread commonality is expressed. A means of measuring the degree of applicability of packages both within and across applications areas is presented. Design considerations that will improve commonality are also discussed. Second, considerations for the development of programming standards are set forth. These considerations will lead to standards that will improve the reusability of Ada packages.

Introduction

By 1990, the cost of software will outpace the cost of hardware by a ratio of five-to-one. According to the United States Department of Defense, the cost of software will rise to $32 billion by 1990, up from $2.5 billion in 1980. The primary responsibility for these high costs can be attributed to the maintenance phase of the software development cycle.

One promising method of reducing these costs and improving the supply is to use what is becoming known as reusable software. Reusable software can be defined to be specifications, designs, data, code, test cases, and documentation that are reused in the same or in a different software program with little or no modifications. Reusability yields a reduction in man-hours required for design, development, testing, and, particularly, maintenance. This reduction in man-hours leads to a reduction in software costs. Since "tried and true" software is used over and over again while bugs are discovered and eradicated, increased reliability is also accrued.

Why hasn't reusable software found widespread acceptance and use by now? The major problem has been the lack of a set of universally accepted standards and a single programming language supporting the design of reusable software. Furthermore, even today, few accept the idea that reusable software could possibly work. Some feel that it is unworkable since a lack of standard and understandable documentation encouraging the use of reusable software exists. Individual company proprietary interests encourage a reluctance to share developed software with other concerns.

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

Frequently, others are reluctant to use software developed for other applications since the software would not serve their current needs.

Portability is, of course, another reason for not accepting the concept of reusable software. Code developed on one machine might not run on another without extensive modifications.

Finally, standards used by one company in the development of their software may differ from another company. This lack of standardization makes it difficult to share software with confidence.

What has occurred to change the picture and begin to turn around the lack of community-wide acceptance of the idea of reusable software? First and foremost, the escalating cost of software is driving the change. As pointed out, these costs are rising to unmanageable proportions! This fact drove the Department of Defense to development and declare the use of Ada as the official programming language for mission critical embedded systems. Reusable software can now be a reality for two reasons: (1) a common language, and (2) a language that supports the theoretical basis for reusability.

We now find both government and private industry seriously considering reusable software systems. For example, the Department of Defense Software Technology for Adaptable Reliable Systems (STARS) is currently working with members of private industry to establish criteria for the design of a reusable software system. Such considerations as the library system approach, parts design, metrics, and incentives for participants are being explored. The output from the team will be a reusability guidebook.

The authors have previously described a reusable software system (Reference 11). Commonality was mentioned as a key element for its design. In Reference 12 some design requirements for commonality were described. This paper now ties together both commonality and standards as considerations for the design of reusable software packages.

Considerations for the Design of Reusable Packages

Regardless of the form that the reusable software system will take, software packages must be designed so that they exhibit certain qualities associated with reusability. If a package is designed with reusability in mind, it will be used again and again. The amount of reuse is a metric that the designer will want to maximize in order to realize the economic advantages of reusable software.

One way of increasing the degree of reuse of software packages is to take specific steps to increase what we call the domain of applicability or the commonality associated with a software package. That is to say, steps must be taken to design software packages that will not only be applicable within a specific applications area, but will also be applicable across applications areas.

Another consideration is to design into the package the basis for reusability and portability. Standards requiring enforcement of these two concepts must be set up a priori to ensure code design that is indeed reusable and portable.

Commonality and standards will now be explored separately to show their importance in the development of reusable software packages.

Commonality

Commonality is two dimensional. Software reuse can be measured by the degree of applicability of the package both within and across applications areas. Applications areas imply distinct industrial groupings. For example, different applications areas could include missiles, aircraft, spacecraft, weapons, ships, lasers, command/control, radar, etc. The economic advantages of reusable software can be diminished if packages developed for a reusable software system do not have the widest range of applicability. If the designer is satisfied with a very narrow range of applicability, or does not consider extending the range of applicability either within the applications area or across applications areas, the reusable software library will begin to bulge with an overabundance of software from a very narrow domain of applicability. Since each package represents development and maintenance costs, it would be economically beneficial to ensure that the designer develop each reusable Ada package with the maximum possible degree of commonality. Furthermore, the proliferation of packages within the reusable software library could create a problem in classification and retrieval of software.

The space shuttle is an example of the non-reuse of software. Of the millions of lines of code developed, not one line was planned for any reuse on any other project. Hopefully, this will not occur for the software developed for the space station. First of all, a common language, Ada, now makes it feasible to develop reusable software. Second, more sensitivity to the need for creating reusable software now exists. However, what is being suggested here is to take a quantum leap in thinking. To develop reusable software within applications areas is not enough even though it would be a step in the right direction. Reusable software that has had every possible bit of commonality designed into it must be developed. This commonality must cross the boundaries of applications areas if we are indeed to reap the economic benefits of reusable software on a large scale.

Increased commonality needs to be a design consideration up front. The designer must consider how to increase the domain of applicability across applications areas. There must be a reluctance to settle for application-specific packages. For example, a program to add two integers together does not have as wide a domain of applicability as an Ada generic package that provides the choice of variable types.

A Commonality Matrix

To place the two dimensions that pertain to the domains of applicability into a visual perspective,the commonality matrix is shown on Figure 1.

WITHIN APPLICATIONS \ ACROSS APPLICATIONS	VERY NARROW	NARROW	WIDE	VERY WIDE
VERY NARROW	0	1	2	3
NARROW	1	2	3	4
WIDE	2	3	4	5
VERY WIDE	3	4	5	6

Figure 1 Commonality Matrix

The domain of applicability is rated from very narrow to very wide in four steps, both across and within applications areas. Software measured against this matrix has a commonality rating from 0 to 6. The higher the rating, the larger the domain of applicability as measured both across and within applications areas.

Software can be classified within the matrix based upon the expected amount of reuse. In order to estimate this, a detailed domain analysis must be performed to identify the possible users within and across all domains.

The objective of the commonality matrix is to identify a point of departure from which steps can be taken to improve and to expand the domain of applicability as an integral part of the design. The first step is to properly classify the software in order to see the possibility of expanding its domain of applicability. If software is thought of as application-specific, such as spacecraft, aircraft, missile, etc., it will be difficult to think in terms of expanding the software's degree of commonality. However, if functions are thought of rather than applications, the range of possible users enlarges. For example, a sort routine for a spacecraft's downlink data also can be used by the accounting industry. In this case, the mind-set should be focused toward the function "sort" rather than the application "spacecraft". Another reason for doing this is to ensure that the library software's classification does not mask the wide range of applications. The sort routine, classified and buried under a spacecraft application would not be discovered by the accounting industry. In this case, the reuse of one sort routine would be diminished while the library would be expanded by another sort routine from the accounting industry. The economic benefits of the reusable software library will decrease!

The economic benefits that can accrue to an industry taking the time, effort, and money to develop truly reusable packages, can be enormous. A spacecraft industry that has developed reusable sort packages can now market its software products in new applications areas!

As an example, assume that a domain analysis of a navigation function within the spacecraft industry resulted in a commonality rating of very wide applicability. On the other hand, considering the applicability of navigation functions to other areas such as aircraft, accounting, etc., a domain analysis resulted in a rating of narrow. The overall rating for this navigation function as evaluated from the commonality matrix would be 4 which is found at the intersection of very wide within the application area and narrow across applications areas. This type of analysis can then be performed with other functions such as math functions, process functions, mission functions, system outputs, and system inputs, etc.

Improving Commonality

The next question that arises is "what can be done to improve the commonality rating of a software package?". A non-reusable package can be thought of as containing application dependent input transformations, application dependent output transformations, and application dependent processes. The package can also contain application independent input and output transformations as well as application independent processes.

One technique would be to create two separate packages. One package would become part of the reusable software library and would contain the application independent input and output transformations as well as the application independent processes and functions. Any transformation or process analyzed to have a narrow range of applicability even within an applications area would be relegated to the non-reusable package. This package would contain application-specific software and would not become part of the reusable software library.

Another technique would be to create an Ada generic package containing the input transformations, output transformations, and processes that have widespread commonality. This package would become part of the reusable software library. An application-specific instantiator would then be written. The function of this package would be to instantiate the Ada generic library package and endow it with all of the application-specific information stored in the instantiator. The instantiator can also be provided with a sequencer in order to instantiate several packages (i.e., input, output, and process).

An example of the first technique is a non-reusable scaler-checker whose function is to take analog and discrete inputs and give messages and scaled data as outputs. The software performs input acquisition, checks for range and limits if the input is analog, checks for desirable states if the input is discrete, scales the inputs, and sends appropriate messages. These functions are supported by a table of ranges, limits, scaling, and messages. By separating the application-specific tables and the conversion to common data types function from the non-application-specific functions performed, a reusable module consisting of range and limit checking, validity checking, and message select functions is formed. The non-reusable module consists of the tables, messages, and conversions to common data types.

Programming Standards

Another consideration for the design of Ada reusable packages is programming standards. In all software development projects, standards are set, documented, implemented by developers, and audited for compliance by a standards auditing team. Typically, programming standards deal with documentation, naming conventions, restricted language statements, anomalies, interfaces, and the like.

If an Ada reusable software library is to be set up, new standards specifically dealing with the design for reusability must be developed. These standards must exist alongside the standards usually written for software development. Each standard must describe a method of implementation that specifically tells the designer or programmer how to comply. Furthermore, a method of compliance control must exist. Compliance control describes the methods that ensure compliance such as automated techniques or auditing procedures.

Many standards are set up merely as guidelines. Typically these standards are not audited. Other standards are set up as mandatory. They must be followed and automated or audited for compliance.

Reusable software will require both standards that heretofore were not a consideration as well as standards that typically have driven software development in the past.

Naturally, reusable software must be be readable and understandable. To ensure this, the source code must follow prescribed templates so that the user will recognize the same format in all packages. Considerations, such as letter case of types, variables, and subprogram names must be established ahead of time.

Standards for formatting must be in place. The reader must see a familiar format from one reusable package to another. Typically for reusable software, information hiding is a requirement. The method of implementation is hidden from the user. This prevents the user from changing the implementation or becoming confused by it. These standards for reusability must apply to the specification part of the package, the part the user will see. Other standards can be set up to deal with the body. For example, such characteristics as indentation, alignment, and spacing must be written. Comments must accompany all code to improve readability.

Typing and declarations must follow a template. Variables should be in a particular order decided upon a priori. For example, all inputs followed by all outputs.

There are many considerations unique to reusable software. It is beyond the scope of this paper to cover all standards required to build reusability into the software. Some of the important ones to be discussed here are:

(1) Accuracy dependency
(2) range dependency
(3) operation order dependency
(4) side effects

Accuracy Dependency
Target machine dependency has an effect upon reusability because of differences in available character sets, differences in exceeding bounds, differences in dynamic allocation and timing effects, the effect upon real-time tasking due to differences in instruction execution time, and differences in accuracy. Factors such as accuracy that affect the portability and reusability of software and should be formalized as programming standards. Floating point operations cannot rely upon the accuracy of the implementation if the code is to be portable. For example, conditional responses cannot rely upon the accuracy of a comparison that can change between implementations. Accuracies must be declared and adhered to. The required accuracy should not exceed that required for a specific application in order to ensure portability to smaller targets. It is a good idea to declare the accuracy of even predefined types to ensure implementation independence. Inequalities using Ada attributes based upon model numbers such as EPSILON, can be used since the same accuracy can be expected with any implementation. Another approach to making "accuracy" implementation independent, is to declare integer and real constants as named numbers of universal type. This leaves it up to the implementation to set the accuracy.

Range Dependency
Range constraints for integer and floating point types should be limited so that the ranges will be independent of implementation. This includes integer literals used for discrete ranges. These literals, unless constrained in the declaration could be out of range on some machines.

The use of attributes that are not model numbers, such as FIRST and LAST, should not be used as a range constraints since these attributes are not implementation independent. Furthermore, values of real types that are outside the range of model numbers cannot be handled by every implementation. Thus if these numbers are used for decisions or exception handling, problems will certainly arise.

It is tempting to handle exceptions by using such declarations as NUMERIC_ERROR and CONSTRAINT_ERROR. Unfortunately, the exact conditions causing these exceptions to be raised depend upon the implementation. Reusability would be better served by programming these exceptions directly into the code.

Order of Evaluation or Elaboration

The order of evaluation of an expression or the order of elaboration of a declaration can be different from implementation to implementation. Standards must be established to ensure that errors do not arise because of differences in the order of elaboration between implementations. The pragma ELABORATE can be used to obtain the same order of elaboration regardless of implementation.

Subexpressions can be evaluated in different ways. Some implementations may evaluate expressions in such a way that causes the subexpression to overflow. Standards must be established to ensure that subexpressions will not overflow under some implementations since range checking cannot be relied upon for intermediate values. One of the ways of accomplishing this task is to limit the number of operators contained in an expression.

Ada generic packages, which will be housed in the reusable library, require special considerations of their own. Code sharing should be avoided. If one package requires code from the other package, the order of compilation will determine if this sharing is possible. Under some implementations this sharing would not even be permitted.

Side Effects

Another consideration in improving reusability and portability, is the elimination of side effects. Side effects are caused by functions that modify variables which are not local to the expression. A reusability problem arises if these non-local variables are used in the function itself. The reason for the problem is simple. The order of evaluation is essential to create the correct value for the function. Since the order changes between implementations, it is unknown whether the value of the variable used in the function was the one before or after the execution of the function. Establishing standards that set forth the order of variable assignment can prevent the problems associated with side effects. For example, if the right hand side of an expression is completely evaluated prior to the assignment to the left hand side, the previous copy of a variable can be relied upon under all implementations.

Summary

This paper described two important considerations for the design of Ada reusable packages: (1) commonality and (2) programming standards. It was shown that reusable packages will bring about economic improvement in software development It is imperative that each reusable package be designed to cover the maximum possible domain of applicability. This maximization implies the designing of the package for applications areas outside of that originally intended. Maximizing commonality can be accomplished by thinking in terms of functions rather than applications areas and partitioning application-specific software from the functions that cut across many applications areas. Developers could realize economic gains by extending software sales outside of their own applications area.

Another consideration of this paper was programming standards. It was shown that many standards, previously not required, must be developed to solve the reusability design problem. The areas of concern covered in this paper was the effect of accuracy, range, order of evaluation, and side effects upon reusability. This does not imply that these are the only considerations. This paper attempted to point the way toward new types of programming standards that will be required for the reusable software of the future.

Acknowledgements
The authors wish to thank Keith Morris for his invaluable input to this paper.

References

1. "Common Ada Missile Packages", Interim Report AFATL-TR-85-17, September 1984-January 1985.

2. Booch, G., "Software Engineering with Ada", The Benjamin Cummings Co., 1983.

3. Freeman, "Reusable Software Engineering:Concepts and Research Directions", ITT Workshop on Reusability in Programming, September 1983

4. Grabow & Noble, "Reusable Software Concepts and Software Development Methodologies", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985.

5. Honeywell, "RaPIER", Final Scientific Report to the Office of Naval Research, Contract No. N0014-85-C-0666, March 28, 1986.

6. Hughes Aircraft Co., "Reusable Software Implementation Technology Reviews", Prepared for NOSC, December 1984.

7. Jones et al, "Issues in Software Reusability", SigAda.

8. McCain, "A Software Development Methodology for Reusable Components", STARS Workshop 1985 Reports.

9. McCain, "Reusable Software Component Construction: A Product-Oriented Paradigm", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985.

10. McNicholl & Anderson, "CAMP Preliminary Technical Report", STARS Workshop 1985 Reports.

11. Nise, Dillehunt, McKay, Kim, Giffin, "A Reusable Software System", AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference, 21-23 October 1985.

12. Nise & Giffin,"The Design for Reusable Software Commonality", DoD STARS Workshop, March 24, 1986.

13. Nissen & Wallis, "Portability and Style in Ada", Cambridge University Press, 1984.

14. Parnas, D., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979.

15. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, December 1972.

16. Procedings of the Workshop on Reusability in Programming", ITT, September 7-9, 1983.

17. "Reference Manual for the Ada Programming Language", MIL-STD 1851A, 22 January 1983.

18. "Reusable Software", Electrical Design News", February 3, 1983.

19. Snodgrass, "Fundamental Technical Issues of Reusing Mission Critical Application Software", STARS Workshop 1985 Reports.

20. "Strategy for a Software Initiative", Appendix II, Department of Defense,1 October 1982.

21. Van Neste, "Ada Coding Standards and Conventions", Journal of Pascal, Ada, & Modula-2, September/October 1985.

22. Wegner, "Capital-Intensive Software Technology", IEEE Software, July 1984.

23. Witte, B., "Checklist for Ada Math Support Priorities", ACM Ada Letters, March, April 1984.

N89-16336

Transparent Ada Rendezvous in a Fault Tolerant Distributed System

Roger Racine
C.S. Draper Laboratory
555 Technology Sq.
Cambridge, MA 02139
(617) 258-2489

Abstract

There are many problems associated with distributing an Ada program over a loosely coupled communication network. Some of these problems involve the various aspects of the distributed rendezvous. The problems addressed in this paper involve supporting the "delay" statement in a selective call and supporting the "else" clause in a selective call. Most of these difficulties are compounded by the need for an efficient communication system. The difficulties are compounded even more by considering the possibility of hardware faults occurring while the program is running. With a hardware fault tolerant computer system, it is possible to design a distribution scheme and communication software which is efficient and allows Ada semantics to be preserved. An Ada design for the communications software of one such system will be presented, including a description of the services provided in the seven layers of an International Standards Organization (ISO) Open System Interconnect (OSI) model communications system. The system capabilities (hardware and software) that allow this communication system will also be described.

Background

There are many reasons for using distributed computer systems. Key among these is the ability to recover when a fault occurs in one of the computing sites. Other reasons include increased throughput and separate subsystem development by different contractors (or the ability to buy off-the-shelf subsystems).

The Ada programming language has the concept of parallelism built in (in the form of tasks). To expand this concept to include running one Ada program on multiple computers, with communication taking place over some network, creates a number of problems. One must consider how to specify the location of processes, the distributed elaboration of the program, whether the various software engineers involved are able to tell where various components will be located, what should happen in the case of hardware faults, and how to implement the various communication mechanisms available in Ada.

In the interest of space, the focus of this paper will be in the area of the distributed Ada rendezvous. In a rendezvous, one task calls an "entry" in another task. The first task then waits for the server task to "accept" the call. Conversely, if the server task attempts to accept the call before it is made, it will wait. When the caller and server both have arrived, the rendezvous occurs, with parameters passed to the entry, a block of code executed, and any output parameters passed back to the caller. The two tasks are then free to execute again in parallel.

That is the simple rendezvous. Ada also provides selective calls and selective accepts, timed calls and timed accepts, and guarded accepts.

A selective call is a call which must be accepted immediately. If any other task is being served, or the server task is anywhere in its execution except waiting at the accept statement, the call is cancelled. The language requires that the server task be checked to determine if the entry is available. It is necessary, therefore, for two messages to be sent over the network to obtain the information. The first message will ask for the rendezvous; the second will either be a message saying the rendezvous could not be accepted or else the second will contain the result of the rendezvous.

A timed call is one which must be accepted within a given amount of time. The call will be cancelled if it is not accepted within that time. The semantics of a timed call are different depending on the value of the delay. If the delay is zero or is negative, the semantics of a selective call will be followed. At least two communication messages must be sent over the network. However, if the delay is positive, and the rendezvous is known, by the caller, not to be able to occur within the delay period, it is not necessary to even attempt the rendezvous. All that is necessary is to wait the delay period before giving control back to the calling task. No communications over the network will be required in this case.

A selective accept allows a server task to accept a call to one entry arbitrarily from among a list.

A timed accept allows a server to wait only a finite time for a task to call one of its entries. At the end of the time period, if no task has called, the server task will regain control, and will execute alternate code.

A guarded accept allows a server to accept, in a selective accept, one of a list of entries based on conditions. The conditions on accepting the various entries will be checked at run-time, and one of the "open" entries will be picked.

The selective accept, the timed accept and the guarded accept can all be managed on the server task's processor, without any network communication.

To a designer of a distributed computer system, these built-in constructs raise a number of issues:

- Should one even allow the use of the Ada constructs when communicating between two tasks on different computers? Specific communication packages could be provided instead, with pragmas used to make the Ada constructs "erroneous". The assumption used here is that the Ada constructs should be used so that the application does not need to know where various tasks are.

- What happens if a task does a timed entry call, but the computer of the called task fails at some time before the rendezvous occurs? It is possible to send enough messages to ensure that the Ada semantics are followed, even in the case of failures, but the time involved in transferring the messages is large. If the rendezvous is extremely inefficient, it is not usable.

- The rendezvous semantics specify that once the rendezvous has started, it must complete before the calling task can continue. What should happen if the processor running the server task fails during the rendezvous?

The AIPS Project

For the Advanced Information Processing System (AIPS), reliability is the most important issue, with efficiency also being a priority issue. The NASA sponsored AIPS project will produce a flexible, fault tolerant, distributed, real-time computer system. It has been designed in terms of "building blocks", such that different applications, such as deep space probe or a manned space station, could use the components.

The building blocks include the following (this is not an exhaustive list. It only includes those blocks pertaining to intertask communication):

- Fault Tolerant Processors (FTPs). One FTP consists of two (duplex) or three (triplex) microprocessors, each executing identical instructions. A triplex FTP has the ability to mask a single fault from the rest of the system. A duplex FTP can determine that a fault exists.

- A fault tolerant Intercomputer (IC) network. This network is a triplicated circuit-switched nodal network with sufficient links in each network to be able to reach all FTPs on the network after experiencing a single fault in the network. Because the network is triplicated, it is possible to have reliable communication with multiple faults.

- Systems software flexible enough to handle an arbitrary number of FTPs connected to the network. The network management process must be able to recognize faults in the network, and reconfigure it automatically (and invisibly to applications processes). The communications software must allow for any number of FTPs to communicate. Systems management software must be able to reconfigure the system (functionally "move" a group of tasks from one FTP to another) if extreme failures occur. Local FTP management software must be able to reconfigure the FTP in the presence of processor failures (downmode to a duplex from a triplex, for example).

- Local Operating System (OS) software capable of working alone (simplex), in duplex or in triplex. The local OS is concerned with the tasks on one FTP (the local scheduler, local rendezvous software, etc.).

Fault Tolerant Distributed Ada

In the absence of fault tolerance, it is difficult to design a rendezvous scheme between tasks on different computers without multiple transmissions over the network to ensure processors remain active throughout the wait for the rendezvous. A message would need to be sent to request a rendezvous. An acknowledgement would be necessary within some time limit in the case of a timed or selective call, to make sure the call has been received and put on the queue. Another message would need to be sent stating that the entry is accepting the call. If the caller is making a timed call, and the delay runs out before this message is received, a message could be sent to take the call off the queue. Finally, the results of the rendezvous can be sent back to the caller.

These messages make up a minimal set of transmissions over the network at the highest level. There might be other transmissions at a lower level to make certain that each complete message is received correctly.

In the fault tolerant AIPS system, the problem of unknown processor failures does not exist. If one of the processors in an FTP fails, the fault is detected. If possible (in a triplex FTP, for example), processing continues normally. If it is not possible to isolate the fault, the System manager will reconfigure such that functions on the failed FTP are run on a different FTP.

For this type of system, it is possible to design an efficient communication service to implement the Ada rendezvous. Because the tasks involved are virtually assured of continuing execution throughout the rendezvous, little error detection needs to be done in the communications software of the processor containing the calling task.

For the case of the timed rendezvous, with a positive delay value, the design calls for the operating system on the called processor to time the wait, if the delay value is larger than the minimum necessary to transmit the rendezvous request and receive a response back. If the entry is not accepted within the given amount of time, a message will be sent back to the calling processor, and the calling task can execute alternate code. The only messages that need to be sent would be the initial message that the caller wants to communicate, and the final message that the server is finished (for whatever reason). If the delay amount is smaller than the minimum needed to transfer messages, no communication is needed. The calling task can be given control back after the specified delay.

If the delay amount is zero or negative, or if it is a conditional call, the messages still must be sent, and the rendezvous might occur.

The IC network services will keep track of whether the called task is moved from one FTP to another. It is also possible that the network will be reconfigured while the tasks are waiting to communicate. All of this will be transparent to the application program.

The design for the intertask communication has been subdivided into two parts, the local communication and the interprocessor communication. The local communication consists of the "normal" rendezvous between two colocated tasks. The interprocessor communication consists of doing the same thing across an IC network.

The "glue" between these two services is called the "context manager". Its function is to determine, for each attempted rendezvous, whether the called entry is on the same processor as the calling task. If it is, the local communication service is invoked. If the called entry is on some other processor, the IC network service is invoked.

The design of the context manager includes a table of locations of what are known as "migratables". As was mentioned above, when a fault is detected, tasks can be transferred to another FTP. The tasks will be grouped into large units. All the tasks within a migratable unit will always be colocated; if they are moved, they will move as a block. Therefore, the table of locations can be organized hierarchically. This will allow a fast algorithm to be designed to determine in which FTP a called task is being run.

The network services are organized into layers, as in the ISO Open Systems Interconnect model. The highest layer, the Application layer, will provide the interface between the context managers on the FTPs, and the IC network.

The interface between the context manager and the IC network has been designed to be as similar as possible to the interface between the context manager and the local communication service. This is not a necessity, but since the context manager is a potential bottleneck, there should be no translation of data to support different interfaces.

The Application layer is responsible for the Ada rendezvous semantics. When a rendezvous is with a task on another FTP, this layer must make sure the semantics are followed. With a fault tolerant system, this layer is fairly simple. At system initialization, a table of task to task communications is used to create logical connections between each pair. When the rendezvous is actually requested by the caller, this layer sends the input parameters, along with the timeout value, to the Application layer on the server's FTP. The server's Application layer calls the server with the appropriate delay (adjusted to take into account communication delays). When the rendezvous is complete, the Application layer returns output parameters. If an exception is raised or the call times out, a message is sent back to the caller specifying the problem. The Application layer on the caller's FTP then either gives control back to the caller at the appropriate point or raises the specified exception to the caller.

The other layers, except the lowest software layer (the Network layer), are designed to support general network services (not just Ada communication), and are not affected by the fault tolerance of the system.

The Presentation layer is responsible for translating data when the format on the receiver is different from that on the sender. The system being built (the proof of concept, or POC, system) has all processing sites identical; therefore no transformation routines will be coded.

The Session layer is responsible for verifying the legitimacy of the communication. It is possible for users (in some anticipated applications) to attempt to communicate with tasks to which they should not be allowed access. A table of allowed communications will be checked for all connections.

The Transport layer is responsible for determining the hardware destination of the communication. It will have a table of locations for the various tasks. If a communication destination is changed (if a task is moved to another processor), this layer will be notified so that communication can continue.

The Network layer is responsible for detecting and masking hardware faults. On a triplex FTP, each processor is connected to one of the three IC networks for transmission. Each processor has receivers all three networks. Masking faults is not trivial when receiving messages from processors which: are not fault tolerant, are duplex FTPs or are triplex FTPs. It is, however, still much faster than detecting faults through multiple acknowledgments at the Application layer. In fact, in the usual case of no faults, a triplex FTP's Network layer needs to do very little processing to obtain (reliable) data for each of the three processors. It is only in the presence of faults that extra processing needs to be done.

The Datalink layer is responsible for sending packets across the network. It contends with the other FTPs for the network using a modified Laning poll which allows one triplex FTP to win the triplicated network in the presence of a single fault. This protocol is somewhat more complex than is necessary for a single network. This added complexity, on the POC, adds a 10% overhead on each transmission. The Datalink layer uses the HDLC protocol to transmit data over each network.

The Hardware layer has two bit-protocols: the data bit and the poll bit.

Conclusion

In the presence of faults on a system which is not fault tolerant, it is difficult to design an efficient communication system to support the Ada rendezvous. For the fault tolerant AIPS computer system, however, it is much easier to design the upper layers of the ISO OSI communications model. The Network layer and the Datalink layer each have more processing to do for each communication, but the amount of processing is small when there are no errors occurring, and the number of communications can be reduced to two at the Application layer.

The result is an extremely reliable, efficient communication system allowing Ada tasks to communicate as if they were on the same FTP.

Distributed systems have many benefits. The distribution allows the system to run in parallel, giving more throughput than in a nondistributed system. The distribution allows the system to be reconfigured in the presence of faults. The distribution allows the system to be able to continue in the presence of damage, by putting the various computers in different parts of the vehicle. Adding hardware fault tolerance complements the distribution by allowing the software to isolate faults and in many cases to mask the fault. This allows software systems such as the communication system to be much simpler than in systems which are not fault tolerant.

N89-16337

Lessons Learned in Creating Spacecraft Computer Systems: Implications for Using Ada^{tm} for the Space Station

by

James E. Tomayko

Senior Computer Scientist

Software Engineering Institute

Carnegie-Mellon University

Pittsburgh, PA 15213*

Abstract

Twenty-five years of spacecraft onboard computer development have resulted in a better understanding of the requirements for effective, efficient, and fault tolerant flight computer systems. Lessons from eight flight programs (Gemini, Apollo, Skylab, Shuttle, Mariner, Viking, Voyager, Galileo) and three research programs (Digital Fly-By-Wire, STAR and the Unified Data System) are useful in projecting the computer hardware configuration of the Space Station and the ways in which the Ada programming language will enhance the development of the necessary software. This paper reviews the evolution of hardware technology, fault protection methods, and software architectures used in space flight in order to provide insight into the pending development of such items for the Space Station.

1. Introduction

During the 25 years since the first flights of manned and unmanned spacecraft carrying onboard computers, the tasks assigned to the machines have grown in complexity and pervasiveness until now it is impossible to consider designing a spacecraft without including substantial computing power. As with any mission critical component, the reliability of computers has to be ensured. NASA's efforts to use computers onboard spacecraft resulted in the development of various methods of fault tolerance. Development of computer systems for unmanned and manned spacecraft have largely followed separate tracks. Systems onboard manned spacecraft used increasing numbers of redundant processors as the primary method of protection. Those on unmanned spacecraft, though redundant, were more innovative in terms of distributing tasks and processing power. The Space Station project provides an opportunity to merge the two tracks, taking from the manned programs experience with using high level languages,

*The author is on leave from The Wichita State University, Wichita, Kan.

This work was partially sponsored by the Department of Defense, with most research done under NASA contract NASW-3714.

software synchronization, and large-scale software development, and from the unmanned programs use of distributed systems and microprocessors. This synthesis creates a system that lends itself to the use of Ada as the onboard software development language if the problems of implementing the language on distributed systems can be solved.

2. A Taxonomy of Spacecraft Computer Systems

A review of previous onboard computer systems is in order to provide a basis for discussing a computer architecture for the Space Station. Since all previous systems have used redundancy in some form for fault tolerance, a taxonomy can be established by considering the nature of the various redundancy schemes. Four types of systems can be identified: simplex, multiplex, functional distribution with full redundancy, and functional distribution with virtual redundancy. Both simplex and multiplex schemes have examples in both the manned and unmanned programs, while the latter pair of types presently have only unmanned spacecraft systems as members.

2.1. Simplex Systems

Simplex onboard computer systems are identified by the absence of redundancy. They are also characterized by being part of a single subsystem of the spacecraft, specifically the guidance and navigation subsystem on manned spacecraft and the commanding subsystem on unmanned spacecraft. If the simplex computer system failed, its tasks would be suspended when possible, or taken over by a backup with reduced functionality. Crew and spacecraft safety would be maintained, but mission objectives would be compromised. Three simplex systems were developed in the 1960s: the programmable sequencer onboard the later Mariner missions, the Gemini Digital Computer and the Apollo Guidance Computer.

2.1.1. Mariner's Programmable Sequencer

Prior to the Mariner Mars 1969 flyby missions, unmanned interplanetary spacecraft carried hardwired sequencers. Essentially these sequencers monitored a counter that was constantly updated by pulses from a clock. When an appropriate time interval had elapsed, some spacecraft activity would be initiated. For example, after the cruise period to a planet, at a time precalculated and put into the sequencer's logic, the spacecraft would orient itself and activate experiments to be done during the encounter with the planet. This meant that very accurate preflight navigation calculations had to be made, and that the sequences could not be changed after liftoff.

Mariner Mars 1969 was to be a double flyby of the Red Planet. If the spacecraft could be fitted with programmable sequencers, then the targeting and camera aiming of the second spacecraft could be changed to follow up on discoveries made by the first flyby. For instance, if a particularly interesting terrain feature was found, the second spacecraft could have its encounter sequence reprogrammed from the earth to obtain more imaging in that area. With a fixed sequencer this would have been impossible. Accordingly, a programmable sequencer with 128 words of memory was included. Later expanded to a 512 word memory, this machine controlled two Mars flyby missions, two orbiters (1971), and the Venus and Mercury flyby mission in 1973. The latter demonstrated the flexibility of the machine because the mission was so complex one software load was too small to do the job. Therefore, a series of complete software loads were prepared and sent up to the spacecraft as the mission progressed [Hooke 1973]. Subsequently, in flight reprogramming became a planned and common feature of interplanetary missions, greatly reducing memory requirements and increasing flexibility.

Backup to the programmable sequencer was the same hardwired sequencer used in the early Mariner missions. If the programmable sequencer had failed, then the mission could continue, but only with preprogrammed sequences. Switching to the backup resulted in reduced functions. A similar situation existed in the two simplex manned spacecraft systems.

2.1.2. The Gemini Digital Computer

The Gemini program was more than a two-man follow-on to the Mercury spacecraft. It was a test bed for guidance and navigation techniques considered essential for the Apollo lunar landing program. Two of the more difficult of these were rendezvous and computer-controlled reentry. A small onboard computer custom-designed and programmed by IBM Corporation provided real-time calculations of maneuvers for the astronauts. During a rendezvous operation, required velocity changes would be displayed and the astronauts would fire thrusters and maintain attitude during powered maneuvers. The spacecraft had lifting capability sufficient to adjust the landing point within a rectangular footprint 500 miles long and 40 miles wide. The computer was programmed to target within the footprint. Each major function was contained in separate single software modules. By using a rotary switch and the start button, a program could be selected. When the machine was not needed, such as during coasting in orbit, it could be shut off.

If the computer failed, its tasks would either be abandoned or done by less effective means. A rendezvous could be canceled. Computer controlled reentry could be replaced by pilot control, such as on the Mercury missions. Either way, crew safety was maintained, but mission objectives were not accomplished.

2.1.3. Apollo's Simplex Systems

NASA contracted with the Instrumentation Laboratory (now the C. Stark Draper Laboratory) of the Massachussetts Institute of Technology for the Apollo guidance system. A computer first built for the Polaris submarine launched ballistic missile was redesigned as the Apollo Guidance Computer. Software for the computer functioned as a priority-interrupt system with some cyclic characteristics. Jobs were scheduled and monitored by an executive program. Code was executed by an interpreter. A typical software load consisted of several dozen "programs" which could be activated by the crew. Key mission phases such as lunar orbit insertion, landing, lunar orbit rendezvous, and entry into the earth's atmosphere were computer intensive activities.

The Apollo was a two-part spacecraft: command module and associated propulsion, and the lunar module. Each module had a computer, with, of course, different applications programs, but the same interpreter and executive. If the command module's computer failed, the mission would be aborted and return to earth would be handled by doing maneuver calculations on the ground and sending instructions to the crew. If the lunar module's computer failed, it had an onboard backup. The backup computer was a small device built by TRW Corporation that could guide the ascent portion of the lunar module to a rendezvous with the command module. That was its sole function, so a computer failure during lunar descent would have caused an abort of the landing attempt.

The obvious shortcoming of simplex systems is that a single computer failure severely damages the mission. This is apparent even in the systems described here, even though their use was limited to one subsystem. As NASA entered the 1970s, spacecraft that depended on computers for more than one function were being designed. On those spacecraft, computer failures would greatly affect crew and spacecraft safety. The first method of reducing the impact of such failures was the development of multiplex systems with full redundancy.

2.2. Multiplex Systems

Three spacecraft designed in the first half of the 1970s used fully redundant computers. The Viking unmanned Mars orbiters and landers and the Skylab orbiting space station both had duplex systems, while the Space Shuttle orbiter and its aircraft predecessor had more than two computers. The introduction of redundancy as a method of fault tolerance necessitated the addition of management software absent from the simplex systems.

2.2.1. Viking and Skylab: Dual Redundancy

The Viking missions to Mars were, in many ways, the most complex unmanned flights yet attempted. A two part spacecraft was placed into Martian orbit, whereupon the orbiter portion began a search for a landing site. When one was chosen, the lander portion descended to the Martian surface. Each part of the spacecraft functioned for years, the orbiter mapping the planet and conducting experiments best done from space, the lander doing chemical and biological analyses of the Martian soil and sending detailed images of the surface back to earth. Both the orbiter and lander had dual computer systems. Each could support its part of the mission independently, or could work cooperatively on separate tasks. The orbiter computers were primarily a replacement for the programmable sequencers carried on Mariners, with the same command and control functions. The lander computer had to control the descent and later the operation of the surface station. The Jet Propulsion Laboratory, which built the orbiter, designed a special purpose processor for its spacecraft. The lander, built by Martin-Marietta Corporation, used an existing Honeywell computer.

Skylab's dual computers were also commercially available, coming from the IBM line of 4Pi processors that were derived from the 360 architecture. The Skylab computers were related to their manned spacecraft predecessors in that they were part of a single subsystem, in this case attitude control. This space station used a complex set of control moment gyros for stabilization and attitude maintenance. The computers were programmed to execute scheduled tasks cyclically, including a set of self-tests. Each cycle the primary computer would deposit a 64-bit status word to a special register in a common section of the system. This register and its associated logic were constructed of triple modular redundant circuits for reliability. If the secondary computer detected that the primary was failing its self-tests, it would take the status word from the common section before the failing computer could corrupt it, and shut down its partner. Such a failure never occurred during the lifetime of the Skylab, but a manual switchover was done to prove that the system was reliable no matter which machine was designated primary.

Even though the Viking and Skylab computers were fully redundant and provided a high degree of reliability, a dual system is insufficient for manned operations. If one half of a dual system detects a failure in the other half, it follows that the failing computer might well detect a failure in the good computer, and will try to shut it down. Also, there is a possibility that the computer detecting the failure is actually the one failing, and that the detection is incorrect. An obvious solution to this dilemma is to add more computers, each running identical software---the solution chosen for the Shuttle.

2.2.2. Redundancy for Digital Fly-by-Wire Aerospacraft

NASA's Space Shuttle is different from all the other spacecraft so far discussed in that the onboard computers have tasks outside of a single system or small set of systems. The Shuttle computers control a large number of spacecraft functions, including such mundane items as the opening and closing of the cargo bay doors. Most importantly, the Shuttle has a digital fly-by-wire control system. This means that where mechanical linkages exist in conventional aircraft control systems, the Shuttle has electrical and electronic connections between the controlling devices, the computers, and the control surfaces. When an astronaut moves the hand controller in the Shuttle, signals are generated and transmitted to the computers, which then generate signals to the actuators at the control surfaces. Therefore, a software or hardware failure makes the control system inoperable, and even a short loss of control in a critical mission phase would be disastrous. Since early research showed that the most likely source of failure in an avionics system would be the computers, NASA chose to increase the levels of redundancy of the primary computer system to provide sufficient protection.

At first, the level of protection was what has been termed "fail-operational/fail-operational/fail-safe." If one computer fails, then the spacecraft is still operational, if a second fails, it is operational, but should return to earth because it has reached fail-safe level, at which another failure would mean serious danger. The fail-safe level escalated to three computers to avoid the stand-off situation. The sum of this is that five computers were necessary and NASA accordingly acquired five IBM AP-101 machines for each orbiter. Later adjustments to the design reduced the level of redundancy to fail-operational/fail-safe, but the fifth computer was kept on the spacecraft as a backup flight system that could be activated by the crew in case of a catastrophic failure of the primary. The backup can only control the ascent and descent of the orbiter, and by itself can not complete a mission.

Of central concern to the Shuttle designers was the development of a redundancy management scheme. Fortunately, NASA was already engaged in a research program that could shed direct light on the subject. The Dryden Flight Research Center at Edwards, California, had been conducting a digital fly-by-wire test program using a modified Vought F-8C aircraft. A single Apollo Guidance Computer was installed in the aircraft to provide flight control. An electronic analog system acted as a backup, but it never was needed. Dryden's research team realized that a simplex system would never be acceptable for routine use, so it was exploring a dual computer system when officials at the Johnson Space Center contacted them about installing three of the same computers to be used in the Shuttle in the F-8 and thus trying out methods of integrating multiple machines into an avionics system. Dryden agreed, and three AP-101s were installed and flown on the F-8. Several single computer failures occurred during flight, none of which endangered the aircraft.

The primary problem in managing multiple computer systems is failure detection. It was reasoned that if the software could be compared at regular intervals, then a failing computer would be obvious because its results would be different from the results of its partners. Comparing checksums consisting of the components of a number of parameters is a simple way of doing this; however, due to differences in the computer clocks, the machines would quickly reach the point where they were out of step, and anomalies would show up in the checksums even though the machines had not failed. To overcome this problem, the machines had to be synchronized. Each time the software executes an input operation, output operation, or changes the module being executed, a three-bit discrete signal is sent on a dedicated bus to the other computers. The sending computer waits up to four milliseconds for its partners to check in with an identical signal. If the signals do not agree, of if the time limit expires, the computer which failed to check in properly is indicated to be failing, and the computer that detects this error goes on. Due to fear of generic errors, the computers are not capable of shutting each other off, only the crew can do that in response to the failure signals.

Basically, multiplex systems provide fault tolerance by layers of redundancy. The disadvantages of this are that entire systems must be replicated at least three times and more reasonably four times to provide the reliability needed by mission critical systems. In a computationally intensive environment, such as that on the proposed Space Station, so many processors would have to be replicated that the increase in power consumption and other resources devoted to the computers would be prohibitive. Other forms of reliability insurance developed for the unmanned flight programs may provide more sensible solutions for the Space Station.

2.3. Functional Distribution with Full Redundancy

NASA's longest lived interplanetary spacecraft are the two Voyagers launched in 1977 and still working successfully, as proved by the recent flyby of Uranus. The Voyagers carry a functionally distributed set of three pairs of redundant computers. Probably most of the reason why this computer configuration was chosen is the structure of the Jet Propulsion Laboratory. Different sections of the Laboratory contribute different components to a spacecraft. In the case of Voyager, the section that builds the command system reused the computer developed for the Viking orbiter with an almost identical software structure. The attitude control system developers used a speeded up version of the command computer and the flight data system had a newly developed machine. Each of the three groups independently determined that the inclusion of a computer system was the best way for the specific tasks involved to be accomplished.

One change caused by adopting functional distribution was the need to communicate with other computers instead of hardwired logic devices. Most intercomputer communication consisted of commands and signals relating to internal tests. More complex communications were required by the next level of unmanned spacecraft systems.

2.4. Functional Distribution with Virtual Redundancy

The next major interplanetary spacecraft designed after Voyager was Galileo, a Jupiter atmospheric probe and orbiter. Galileo carries a dual computer system for attitude control and pointing that uses an off-the-shelf microprocessor, the ATAC-16, and is programmed in a high level language, HAL/S. Its command and data system also uses commerically available microprocessors, six RCA 1802s in two strings of three. This system was derived from research sponsored by the Jet Propulsion Laboratory concerning reliable computer systems for unmanned spacecraft.

Beginning in the early 1960s, the Laboratory sponsored the design of a computer called STAR (for *S*elf *T*esting *and* *R*epair) that consisted of collections of multiple copies of each major component [Avizienis 1968]. For instance, memories, input/output devices and the like were triplicated. A special piece of hardware called the Test and Repair Processor, or TARP, had five copies. When the computer was operating, one of each subcomponent and three TARPs were powered up and running. If the TARPs voted that a component was failed, they activated one of the spares. If the vote had not been unanimous, the dissident TARP would be shut off and another activated. In this way no more than the minimum number of components would be powered at any given time. The weakness of this scheme is that a failure of the switches used to turn off bad components and turn on good ones would negate the fault tolerance. However, the concept of a single computer with virtual redundancy survived to the next round of research.

Research initiated after STAR led to the construction of the Unified Data System in the early to mid 1970s [Rennels 1978]. Here the emphasis was on several processors working cooperatively. Certain processors, called High Level Modules, would communicate only with other processors, called Terminal Modules. The Terminal Modules would deal with spacecraft systems or the outside world. Conceptually, by carrying several High Level and a larger number of Terminal Modules, each communicating by means of multiple busses, and sharing numerous memory modules, the system could function with a variety of combinations of modules, memory, and bus connections. This way a single processor failure would result in a change of the configuration, but no degradation of performance unless a number of different failures occurred.

Designers of the Galileo command and data system did not fully adopt the concept, even though they adapted the terminology. Two 1802s are assigned as High Level Modules, four are Low Level Modules. Several memories and redundant busses are part of the system. However, it is fundamentally separated into two redundant strings. Even so, the software is constructed in what are termed "virtual machines" and is distributed over the several processors. From the Unified Data System and the actual Galileo software some hints for a possible Space Station computer architecture can be derived.

3. Computer Architecture for the Space Station

The Space Station will be different from any previous manned spacecraft in terms of its computational needs. In fact, it will be much closer to an unmanned spacecraft. This is primarily because the guidance and navigation tasks on a Space Station are minimal compared to what a spacecraft like the Shuttle requires for active flight control. However, considerable computational capability in the areas of data acquisition and analysis, attitude control, life support, and spacecraft health monitoring will be necessary. With this variety of tasks, it is logical to imagine that the final configuration of the computers onboard the Space Station will be a distributed system, with physical processors embedded in the hardware built to accomplish each function. Thus the Station's computer systems will resemble the functional distributions used on Voyager and Galileo, rather than the centralized systems used on the Shuttle.[**] Questions of redundancy can then be handled at the local level. Some systems such as life support are so critical as to require fault protection to the same degree as flight control, and will require multiple dedicated processors for redundancy. Other systems can be virtually redundant in that their tasks can be transferred to another processor in another system in case of a failure. Perhaps a common pool of processors can be made available to host tasks offloaded from failed machines. In any case, the intent of a hardware

[**]Although the Shuttle has "local" computers on the main engines and on payloads, the Data Processing System, with its multiplex configuration, does all other computational operations.

architecture for the Space Station should be to provide fault tolerance relative to the importance of the systems, and to avoid carrying large numbers of resource-hungry multiplex systems.

4. Implementing Space Station Onboard Software with Ada

Since Ada has already been designated as the development language for the Space Station, its strengths and weaknesses in implementing software for its potential computer architecture are of interest. Ada's strengths in developing this type of system lie in its inherent ability to handle concurrency, both in terms of data sharing and synchronization, and in hiding mechanisms of concurrency in the programming language. Fundamentally, the entire software load for the Station could be created as a set of tasks, some of which will run individually on separate processors, some of which will share a single processor, but all of which can be considered as part of a library of related programs. This was impossible in previous distributed systems in which the software for each computer was written separately in different languages, sometimes in a mix of high level and low level languages, and interconnected with great difficulty. The chief weakness of Ada at this point in time is not the language itself, but the lack of implementations of it that make use of its full range of features, particularly those most applicable to the Space Station.

4.1. Ada Features Most Useful for Distribution and Fault Tolerance

Since the original purpose of designing Ada was to serve the development of large and real time systems, several features of the language are directly applicable to programming the heterogenous machines on the Space Station.

4.1.1. Tasks

Using Ada, programs can be made up of a variety of units, including *tasks*. A task is a program unit that runs in parallel to other tasks, and to the main program, which is implicitly also a task. Moreover, it can run either interleaved with other tasks in one physical processor or as a single process on a machine in a multicomputer system. Tasks on the Space Station would have varying degrees of interaction. For instance, a task monitoring spacecraft health would periodically wish to receive signals from processes throughout the Station in order to make sure everything is still functioning. These messages would be far less frequent than three computers running identical tasks as part of a mission critical, locally redundant, synchronized subsystem. Regardless of the level of communication, the information to be exchanged can be abstracted in the task body, hiding the complexity of the interior of a task from programmers working on associated tasks.

4.1.2. Rendezvous

Previous parallel computations in spacecraft shared information by message passing or common data pools. On Voyager, messages are sent between the command computer and attitude control computer as single units. On the Shuttle, the high level language HAL/S provides for the declaration of common data shared by several scheduled parallel processes. Ada provides for common data using pragmas for shared information, but the most common form of information exchange on the Station would probably be message passing, usable for simple data exchange or for synchronization. Message passing is implemented in Ada using the rendezvous, in which a task will be blocked while attempting to send or receive a message. When both sender and receiver reach the point in their respective task bodies where they are ready to do the exchange, a rendezvous occurs, data is transferred, and both tasks continue. These rendezvous can take place between widely distributed tasks.

4.1.3. Exceptions

One Ada feature critical for Space Station systems is the ability to gracefully handle predictable errors. Even though most Space Station subsystems could have short duration failures without endangering the crew, actively handling the failures as opposed to reacting to existing conditions is almost always preferable. Exception handlers can be part of each task, and, used creatively, can eliminate complete shutdowns of subsystems.

4.1.4. Modularity

Since the Space Station is expected to operate over a long period of time, with many changes in its component modules, the software used on it must be easily modifiable. Ada's ability to separately compile tasks that have been added or modified and include them in the existing software load is a significant advantage. NASA has made good progress in reusing software in preparing Shuttle flight loads. Consciousness of reusability can be easily transferred to the Space Station project since the development language directly supports such techniques through the use of generics.

4.2. An Example: Implementing Shuttle-Like Computer Failure Detection in Ada

As an example of tasks, rendezvous, and exception handling, the Ada code in Figure 1 on the next page implements the Shuttle computer failure detection and synchronization scheme in a two processor system.

5. Summary

Ada has many characteristics that support the development of software that implements fault tolerance schemes developed for previous spacecraft. Also, the ability to run on distributed systems essentially transparently to programmers working on the Space Station software means that a variety of redundancy configurations can be used. This, of course, depends on continued research and development concerning implementing Ada. Although some attempts have been made to implement Ada on several cooperating processors, the nuances of doing so are still not all understood [Ardo 1984]. Also, the progress of Ada development environments, though picking up steam, is still behind original expectations. Adoption of existing Ada development technology by the Space Station project coupled with support of efforts designed at multiprocessor implementations provides the safest route to completion of Space Station software in the early 1990s.

```
-- The task specification states which entry points and parameters
-- will be honored by the task.

task SYNCHRONIZE is
   entry IO_EVENT (I_OR_O: in OPERATION_TYPE);
          -- where I_OR_O is a value indicating whether the sending task
          -- running on the host processor has just done an input or output.

   entry SYNCH_SIGNAL (SIGNAL: in OPERATION_TYPE);
          -- where SIGNAL is a value indicating whether the sending task
          -- located on the other processor has just done an input or output.
end SYNCHRONIZE;


task body SYNCHRONIZE is

   WAIT_INTERVAL: constant DURATION := MILLISECONDS*4.0;

begin
 loop
        -- this task activates in parallel to the others and runs forever.

  begin
     accept  IO_EVENT (I_OR_O: in OPERATION_TYPE) do
        -- the task is blocked at this statement waiting for one of its
        -- fellow tasks on this processor to send word that it has done
        -- an input or an output. When a value for I_OR_O arrives, then
        -- it waits for up to 4 milliseconds for the other processor to
        -- check in using this select statement:

        select
               accept SYNCH_SIGNAL ( SIGNAL: in OPERATION_TYPE) do

                  -- check to see if the signals match, if not,
                  -- raise an exception.

                       if I_OR_O /= SIGNAL then
                          raise OUT_OF_SYNCH;
                       else
                          --continue
                       end if;
               end SYNCH_SIGNAL;

        or

               delay WAIT_INTERVAL;

               raise TIME_EXPIRED;

        end select;
      end IO_EVENT;

    exception
      when OUT_OF_SYNCH =>
         -- handle exception caused by a mismatch of action codes.

      when TIME_EXPIRED =>
         -- handle exception caused by a failure to synchronize in time.
  end;
 end loop;

end SYNCHRONIZE;
```

Figure 1

Bibliography

[Ada 1983]*Ada Programming Language*, ANSI/MIL-STD-1815A, 22 January 1983.

[Ardo 1984] A. Ardo, "Experimental Implementation of an Ada Tasking Run-time System on the Multiprocessor Computer Cm*", *Proceedings of the First Annual Washington Ada Symposium.*

[Avizienis 1968] A. Avizienis, "An Experimental Self-Repairing Computer," NASA-TR-32-1356, Jet Propulsion Laboratory.

[Gehani 1983] Narain Gehani, *Ada, An Advanced Introduction* (Prentice Hall).

[Habermann 1983] A. Nico Habermann and Dewayne E. Perry, *Ada for Experienced Programmers* (Addison-Wesley).

[Hooke 1976] A. J. Hooke, "In Flight Utilization of the Mariner 10 Spacecraft Computer," in *Journal of the British Interplanetary Society,* Vol. 29, April, 1976.

[Rennels 1978] David A. Rennels, "Reconfigurable Modular Computer Networks for Spacecraft On-board Processing," *Computer,* July, 1978.

Notes on the Author

James E. Tomayko is a Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, a federally-funded research and development center. He is on leave from a faculty position in the Computer Science Department of The Wichita State University. Recently Dr. Tomayko completed a three-year study of NASA's use of computers in space flight operations to be released by the Agency as a book. Articles related to the subject of the present paper have already appeared:

- "NASA's Manned Spacecraft Computers," *Annals of the History of Computing*, Volume 7, #1, January 1985, pp. 7-18.
- "Achieving Reliability: The Evolution of Redundancy in American Manned Spacecraft Computers," *Journal of the British Interplanetary Society*, Volume 38, #12, December, 1985, pp. 545-553.
- "Digital Fly-By-Wire: A Case of Bidirectional Technology Transfer," *Aerospace Historian*, Volume 33, #1.

Dr. Tomayko is a National Lecturer for the Association of Computing Machinery, and has given over 50 talks on manned and unmanned spacecraft computer systems within the last 18 months.

N89-16338

USING ADA* -- THE DEEPER CHALLANGES

David A. Feinberg, C.D.P.

Software Technology
Boeing Aerospace Company
P. O. Box 3999, M/S 82-53
Seattle, Washington 98124
Telephone: (206) 773-5485

ABSTRACT:

The Ada programming language and the associated Ada Programming Support Environment (APSE) and Ada Run Time Environment (ARTE) provide the potential for significant life-cycle cost reductions in computer software development and maintenance activities. The Ada programming language itself is standardized, trademarked and controlled via formal validation procedures. Though compilers are not yet as production-ready as most would desire, the technology for constructing them is sufficiently well known and understood that time and money should suffice to correct current deficiencies.

The APSE and ARTE are, on the other hand, significantly newer issues within most software development and maintenance efforts. Currently, APSE and ARTE are highly dependent on differing implementer concepts, strategies and market objectives. Complex and sophisticated mission-critical computing systems require the use of a complete Ada-based capability, not just the programming language itself; yet the range of APSE and ARTE features which must actually be utilized can vary significantly from one system to another. As a consequence, the need to understand, objectively evaluate, and select differing APSE and ARTE capabilities and features is critical to the effective use of Ada and the life-cycle efficiencies it is intended to promote. Methodologies for dealing with dissimilar APSE/ARTE systems are also in sore need of definition and understanding; particularly for industry contractors who will be developing similar capabilities (e.g., missile and air/space craft navigation, guidance, throttle control) for differing customers (e.g., Army, Navy, Air Force, NASA, Boeing, Airbus).

It is the selection, collection, and understanding of APSE and ARTE which provide the deeper challanges of using Ada for

* Ada is a registered trademark of the United States Government (Ada Joint Program Office)

real-life mission-critical computing systems. This paper discusses some of the current issues which must be clarified, often on a case-by-case basis, in order to sucessfully realize the full capabilities of Ada.

1. INTRODUCTION

In the early 1970's, the Department of Defense (DOD) recognized several problems related to the acquisition of software for major defense systems. Software systems were too frequently late, unreliable, and more expensive than planned. Additionally, there was a steadily rising trend in software costs while, at the same time, computer hardware costs were decreasing significantly.

At the time, the primary cause of these problems was identified as a deficiency in the computer programming process; particularly in the area of programming languages. There were over 450 general purpose languages and dialects being used for DOD systems with no single point of control for each. Many of these languages were poorly suited to their application, and/or did not take advantage of nor support good programming practices. The DOD was also beginning to recognize the long-term life-cycle advantages of using higher order languages (HOL's) rather than assembler code. By 1974, each of the military services was independently proposing development of a standard HOL for their service's mission-critical software development.

In January, 1975, a joint services HOL working group began identifying and defining requirements for all DOD HOL's and individual service efforts were halted. The "Strawman" document issued in April, 1975, started a multi-year effort which culminated in 1981 and 1983 with the establishment of ANSI/MIL-STD-1815A, "Reference Manual for the Ada Programming Language," as a single DOD standard for all future mission-critical computer software development efforts.

Unfortunately, during the six years required to produce the Ada standard, the understanding of the problems of developing large, complex software systems evolved. While the programming process was still important, newer full-life-cycle models of software project activities reduced programming's overall significance to only 20% of the whole; much less than was thought in the early 1970's.

In response to this changing perception, the HOL working group began to recognize that the new common DOD HOL alone would not be sufficient to ensure DOD's desired improvements in software development. The programming environment within which Ada would operate needed significant improvement.

Following two years of work, an Ada Programming Support Environment (APSE) was defined in the 1980 "Stoneman" document. Even though this document provides criteria for assessment and evaluations of programming environments, it is not a standard and, as such, implementers of Ada tools are not bound by any hard and fast requirements. Rather, implementers are free to choose any of the four "Stoneman"-defined levels of Ada programming support. More importantly, they are also free to select, as they see fit, specific tools within each of the levels. Thus, while Ada, the language, is tightly controlled, APSE's are not controlled at all and vary significantly from one implementer's products to another's.

In a similar manner, an Ada Run Time Environment (ARTE) can also vary significantly. Once the necessities of the Ada language standard are satisfied, implementers are free to produce a wide variety of operating executives. In fact, ARTE development is even less constrained than development of an APSE; no assessment and evaluation document such as "Stoneman" even exists for run time requirements.

In response to the absence of APSE and ARTE system standardization, projects using Ada must, on a case-by-case basis, identify those features most necessary to their specific requirements. Once this is done, evaluation of the numerous implementer offerings is required in order to select the critical environmental capabilities which will be used. The following sections describe the key issues affecting selection of Ada Programming Support and Ada Run Time environments.

2. ADA PROGRAMMING SUPPORT ENVIRONMENT

An Ada Programming Support Environment (APSE) consists of a number of individual tools which provide software support to write, test and maintain Ada language programs. An APSE can also be used to provide orderly program development methodology. Tools within an APSE will vary from implementer to implementer; however, most implementers conform at some level to the "Stoneman" document. The cooperating ability of tools with each other, as opposed to merely "Stoneman" tools-database interfaces, can, however, vary significantly.

Typically, an APSE will consist of at least the minimum tool levels described in "Stoneman": an operating system, a Kernal APSE (KAPSE), and a Minimal APSE (MAPSE). With the exception of a debugger, it is virtually impossible to utilize Ada without the MAPSE tools: a compiler, linker/loader, editor, configuration manager, and job control language processor.

Additionally, a full APSE (i.e., any Ada Programming Support Environment with tools in excess of those called for by MAPSE) may consist of any number of augmenting tools such as a pretty printer, cross reference generator, test generator, program design language processor, source code control system, problem reporting system, etc.

Evaluation of an APSE is required in order to determine which available environment best fits the needs of a specific Ada-based project. This minimally requires analyzing the tools in a given APSE to determine their effectiveness, and where possible, to directly compare them to similar tools in other APSE's.

While quantitative methods can be used to examine many tools, this is not always possible. First, even though two (or more) tools perform the same function on the same computer using the same operating system, their performance characteristics may vary significantly based on computer load factors at the time of testing. Even if these factors can be controlled or mitigated, design parameters of the tools themselves can cause fluctuating performance data depending on individual account and session situations. In general, modern virtual memory multi-component computer systems can play havoc with what appear to be straight-forward quantitative evaluations.

The second reason is that quantitative evaluation methods are not always applicable. Discussions of such factors as "user friendliness" do not realistically lend themselves to quantitative accumulation. Even so, these factors can be significant issues when determining the overall effectiveness of a tool.

While individual tool evaluations are important, even more critical is extending any evaluation to the integration and cooperation of all of the tools which comprise an APSE. It is not uncommon for individual software tools to be efficacious as stand-alone entities, yet efforts to use the results of one as grist for another fail totally. Such an overall view of APSE effectivity and suitability cannot be obtained by simply summing the results of individual tool evaluations. An APSE must be reviewed as an integrated (or non-integrated) whole to determine if it fulfills a project's software development needs.

3. ADA RUN TIME ENVIRONMENT

An Ada Run Time Environment (ARTE) is the collaboration of program object code conventions with data structures used to interface to the underlying run time system. This system, in turn, consists of a series of library and/or executive routines that are necessary to support execution of Ada programs. Typical functions of an ARTE include general operating system services as well as Ada-specific features such as tasking, dynamic memory management, exception handling, interrupt processing and any other needed support deferred from a compiler's code generation phases.

Even though the Ada language is standardized, the ARTE for different computers and operating systems can vary widely. This can be due to differences in computer hardware, operating systems, compiler implementations of Ada semantics, or, the most frequent case, a combination of all of these. Additional variations can result from trade-offs for reasons of ARTE or program size, speed, overhead, capability, or portability.

In rare cases, a specific project using Ada will find one or more ARTE implementations which are universally best suited to its needs. Usually, however, compromises between various implementations in terms of project priorities will be required. Given the characteristics of most mission-critical software programs, the best ARTE may turn out to be the one that is easiest and safest to modify on a case-by-case basis.

Evaluation of ARTE elements depends on the depth to which a project is required to delve. Some elements (e.g., code size, coding language, implemented pragmas) are readily apparent by simple examination of external characteristics or implementer documentation. Others (e.g., subprogram call timing, arithmetic implementations) can be found through test program executions. Still others (e.g., delay overhead, task dispatch algorithm) can only be determined by detailed analysis (or even experimental modifications) of the run time code itself.

4. ENVIRONMENTAL PROLIFERATION

Even though the Ada programming language itself is standardized, trademarked, and controlled via formal validation procedures, Ada Programming Support Environments (APSE) and Ada Run Time Environments (ARTE) are not. The U. S. Army has already taken delivery of its APSE/ARTE system: the Ada Language System (ALS). The Air Force continues to make progress on key components of its support environment: the Ada Integrated Environment (AIE) and its supporting Ada Compilation System (ACS). Within the past few months, the Navy has let a contract for its version of the ALS: ALS/N. NASA has also established its policy calling for an integrated Software Support Environment to support use of Ada for Space Station operational software.

Thus far, with the partial exception of ALS and ALS/N, none of the existing APSE/ARTE systems are compatible with each other; even though they execute on identical host and target computers. When systems on the drawing boards plus commercially available products (e.g., Systems Designers' "Perspective", Verdix's "VADS") are added to the list, the proliferation of dissimilar capabilities, facilities and functions will reach significant proportions. The late 1980's have all the potential to become highly reminiscent of the 1970's programming language proliferation which led to Ada in the first place (Figure 1).

ORGANIZATION	1970'S HOL	1980'S APSE
Air Force	JOVIAL	AIE/ACS
Army	TACPOL	ALS
Navy	CMS-2	ALS/N
NASA	HAL/S	SSE
Industry	FORTRAN Pascal C	Perspective VADS ADE etc.

Figure 1. Organizational Standards Proliferation

The potential proliferations in late 1980's APSE/ARTE are the well-intentioned result of attempts to "graft" enhancements onto the Ada programming language, which is in turn, the solution to the 1970's perception of the software development problem. Ada was initially designed to correct difficulties in programming. Current, 1980's, estimates allot only 20% of the software development cycle to programming, and consequently, Ada needed to be expanded to fit a newer, better, full-life-cycle model. Unfortunately, the "Stoneman" grafts have been done "on-the-fly", and have, in turn, recreated a mutant of the initial problem. The Ada Language is standardized. APSE´s and ARTE´s are not. Moreover, differing organizations are beginning to require use of their incompatible APSE/ARTE even as full-life-cycle model methodologies for software development are beginning to coalesce (e.g., DOD-STD-2167).

5. CONCLUSION

The use of Ada and its associated Ada Programming Support Environment (APSE) and Ada Run Time Environment (ARTE) continues to provide a high potential for significant life-cycle methodology improvements and cost reductions in software development and maintenance activities. In order to move the significant advantages of Ada from potential to actual, several concurrent efforts must be completed. The first, development of high quality compilers and optimizing code generators, is already well underway. Over a dozen organizations currently offer Ada compilers and some form of minimal programming support tools. The technology necessary to improve these offerings has been in existence for over a dozen years. Time and incentive should produce the needed production quality compilers.

Development of full-function, integrated, APSE´s is the second needed effort. While the goal of this effort is conceptually clear, the steps necessary to reach it remain unacceptably vague. Full scale software development environments have been proposed for years, but no universally usable one yet exists. Using Ada as a vehicle for producing such a capability has much merit and the "Stoneman" document provides some necessary guidance. Unfortunately, these items are not yet enough. Significant research into programming environment requirements and solution sets, particularly those dealing with human factors and expert systems, remains to be accomplished.

The third effort needed to move Ada from potential to actual usage is the development of a configurable ARTE. Ada is intended for "mission-critical" computing systems. These systems can range from ground-based surveillance and tracking systems (air, space, sea) to in-flight avionics (manned, unmanned) to simple sensor/actuator systems, and much much more. Even though all of these mission critical systems can be considered as "real time," many other widely varying characteristics can affect their execution environment constraints. A great deal of research and development remains to be done. The need for an ARTE criteria and evaluation document is barely even recognized. Yet, the ultimate key to mission-critical computing is its performance in the field; under "production" conditions.

Finally, and most difficult, is the need to recognize and begin to resolve the issue of incompatible APSE/ARTE systems. Using the full-life-cycle model demanded of today's software development process, the proliferation of differing services' tool sets can clearly become counter-productive; particularly for organizations performing similar work for different customers.

The work of many individuals and organizations will be required to complete the efforts described in this paper. The definition, rationalization, implementation and integration of APSE and ARTE into the Ada language to create complete software development environments are now the deeper challanges of using Ada. Only when they are accomplished will Ada be able to meet the ultimate goals for which it was created.

ACKNOWLEDGEMENTS:

The work of many individuals is necessary not only to answer the questions raised in this paper, but to raise and clarify them in the first place. Several members of Boeing's Ada Project have contributed ideas and concepts which led to this paper. Special recognition belongs to two: James B. Unkefer for his work on Ada Programming Support Environments, and Ruth A. Maule for her clear, effective approach to the enigmas of Ada Run Time Environments. Thanks are also due to Maretta Holden of Boeing Military Airplane Company who continues to see into the future farther than most of us.

REFERENCES:

1. AJPO, "Kernal Ada Programming Support Environment (KAPSE) Interface Team Public Report," Volumes I-V.

2. ARTEWG, "Draft Charter for the Ada Runtime Environment Working Group," July, 17, 1985.

3. ARTEWG, "Ada Implementation Dependencies," November 12, 1985 (draft).

4. United States Air Force, "Preliminary Program Manager's Guide to Ada," document numbers ESD-TR-83-255 and WP-25012, February, 1984.

5. United States Department of Defense, "Ada Methodologies: Concepts and Requirements (METHODMAN)," November, 1982.

6. United States Department of Defense, "Interim DoD Policy on Computer Programming Languages," Memorandum to Secretaries of the Military Departments, et. al., from Under Secretary of Defense Robert DeLauer, June 10, 1983.

7. United States Department of Defense, "Proposed Military Standard Common APSE Interface Set (CAIS), Version 1.4," October 31, 1984.

8. United States Department of Defense, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A, February, 1983.

9. United States Department of Defense, "Requirements for Ada Programming Support Environments (STONEMAN)," February, 1980.

BIOGRAPHY:

David A. Feinberg, C.D.P., is a specialist in the development and use of software engineering tools and environments. He is employed by The Boeing Company and is currently in charge of the company's Ada Project. During the past twenty-three years, Mr. Feinberg's assignments have included creation of a software development facility used for the construction of commerical electric power distribution and control products; large scale network operations and communications management; and compiler and operating systems construction. He is the author of over twenty-five papers, essays and articles. Mr. Feinberg is a member of ACM, IEEE Computer Society and DPMA, and holds an M.S.A. degree from The George Washington University and a B.S. degree from Stanford.

N89-16339

AN ADA IMPLEMENTATION FOR FAULT DETECTION, ISOLATION AND RECONFIGURATION USING A FAULT-TOLERANT PROCESSOR

Gregory L. Greeley

The Charles Stark Draper Laboratory
555 Technology Square
Cambridge, Massachusetts, 02139 USA
(617) 258-2482

Abstract

This paper covers the design and implementation, in Ada, of the Fault Detection, Isolation and Reconfiguration (FDIR) Manager for the triply redundant, tightly synchronized, Fault Tolerant Processor (FTP). It also examines the suitability of Ada, in the context of the FTP, for real time control tasks. This paper explains the operational concepts behind the FTP, and discusses the structure of the resultant Ada code.

This work is supported by NASA under JSC contract NAS9-17560.

1. Draper Laboratory's Fault Tolerant Processor

1.1 Background

In April of 1983, the Charles Stark Draper Laboratory undertook the design and construction of a "distributed, fault and damage tolerant, real time information processing system" for aerospace vehicle control [2] [1]. This proof-of-concept system is known as the Advanced Integrated Processing System (AIPS). The goal of the AIPS project is to make a fault tolerant network of fault tolerant computers behave as a single highly reliable system. The AIPS system is composed of several Fault Tolerant Processors that are linked together via two networks: an inter-computer (IC) network, and an input-output (I/O) network.

The inter-computer network is used for communication among the FTP's. This network allows the FTP's to coordinate their actions and the division of tasks. The IC network is also used to report errors and failure conditions. The I/O network carries all input to and output from the AIPS FTP's. Thus, all sensors and actuators may be accessed by any FTP, and since FTP's are not tied to specific I/O devices, any FTP may run any I/O dependent task. This flexibility was built into AIPS so that tasks can "migrate" between FTP's without concern about which specific I/O devices are attached to the individual FTP's.

This paper concentrates on failure detection in the local FTP's, and further discussion on the operation of these two networks is beyond the scope of this paper.

1.2 The Fault Tolerant Processor

The AIPS Fault Tolerant Processor achieves a high level of reliability by using three identical processing elements that perform identical operations on identical input. The FTP will continue to operate correctly even after the failure of one of its channels, because data from the two good channels will vote out and mask data from the faulty one. The design goal of the FTP is to produce a fault tolerant virtual processor out of these three tightly synchronized channels. Thus, the programmer who writes applications for the FTP does not have to worry about the fact that there are actually three processing units that are continually voting all input and output. In the Draper Fault Tolerant Processor, specialized hardware maintains synchronization and handles communication between processing sites. This solution not only reduces the software overhead, but, in fact, allows the FTP to be treated as a virtual processor. Because none of the instructions in the user's application software reveal the fact that the FTP is actually three processing units, it is hoped that this virtual processor abstraction will reduce software cost and complexity in fault tolerant systems.

Data exchanges, which are necessary both for communicating with the other channels and for voting, are done by the hardware data exchange mechanism. Data is voted on a bit by bit basis: the hardware compares each set of three bits and masks out any bit that disagrees with the other two. If an error is detected, a hardware error latch is set, noting the type of exchange and the channel(s) at fault. Fault detection is implemented by comparing the voters' inputs and outputs; fault isolation uses the pattern of errors latched by the voters. By supplying this fault detection and masking in hardware, the FTP frees the software of this burden and helps provide the virtual processor abstraction. These concepts of hardware implemented fault tolerance and data exchanges have been successfully demonstrated in the Fault Tolerant Multi-Processor [4] at Draper Laboratory, and the theoretical basis for this interconnection scheme's protection against Byzantine failures can be found in [7].

ORIGINAL PAGE IS
OF POOR QUALITY

1.3 Data Exchange Mechanism

The data exchange mechanism is the FTP's primary means of correcting for failures. It has been shown [6] that Triple Module Redundant (TMR) systems such as the FTP need two basic types of data exchanges: a triplication and a direct vote. A triplication is used in the case where a single channel has a local value, such as a sensor reading or keyboard input, that must be sent to the other two channels. Since a direct transmission's reliability is vulnerable to a single point failure, the triplications are sent through the voters. A direct vote, on the other hand, is used in the case where three channels have computed identical outputs, such as actuator commands or terminal output, that must be voted to correct for errors before transmission.

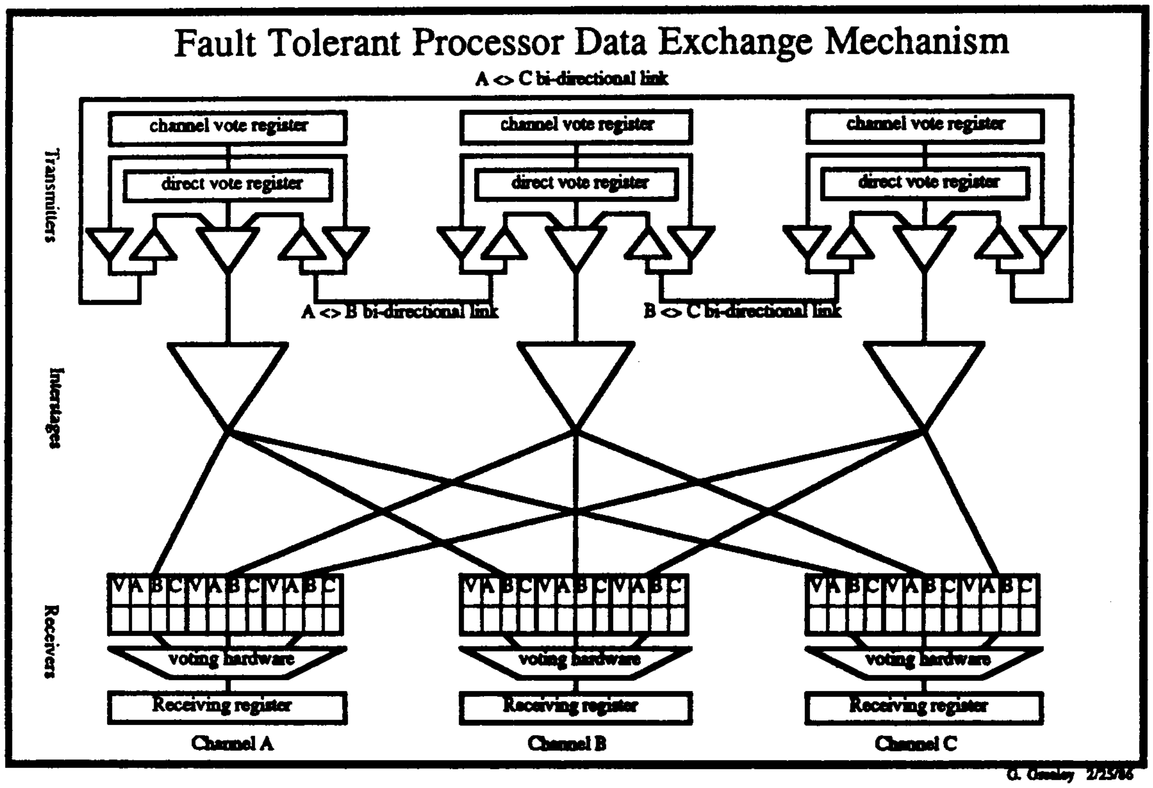


Figure 1-1: FTP Data Exchange Mechanism

Figure 1-1 shows a schematic representation of the FTP's data exchange mechanism. Note that there are three major elements in the mechanism: the transmitters, the interstages, and the receivers. These elements are connected in several different ways. First, each channel's transmitter has a bi-directional link to the other two channel's transmitters. These links are used for immediate access to raw data during triplication data exchanges. Second, each transmitter has a link to its interstage. This link is used to send data to be latched by the interstage for further re-transmission. Finally, each interstage has a link to each channel's receiver. These links are used by the interstages to send a copy of their data to each channel.

During an exchange, each of the elements in the data exchange mechanism has a different function. The transmitters must configure their data paths so that the correct data is sent to the interstages. Each transmitter may send either its own data or the data available on one of the direct links from the other channels. The interstages must latch the data, triplicate it, and send a separate copy to each of the three receivers. Finally, the receivers are responsible for latching and voting the three copies of the data from the interstages. The bit by bit majority vote is done in hardware, and the result will be stored in the receiver register. If there are any disagreements in the voting, they are recorded in the voter's error latches.

Each channel's receiver has a 12-bit dedicated error latch. These twelve bits are divided into three sets of four bits. Each set is used to record errors from a specific channel, and each bit within a set is used to specify what type of exchange the error occurred in: direct vote or triplicating from A, from B, or from C. Thus, if channel A's voter discovers a disagreement in channel B's value while triplicating a value from C, it will set a specific bit for that exchange in its error latch. As more errors are discovered, more bits will be set, but none will be reset. Only a specific command from the software can reset the bits in the error latches.

In their well-known paper on the Byzantine General's problem [5] Lamport *et al* show that three processors (meaning three fault containment regions) cannot reach agreement in the presence of a fault. To surmount this problem, the FTP is divided into six fault containment regions: the three channels and the three interstages. That is, each channel and interstage is isolated (physically and electrically) so that a fault in one cannot cause a fault in another. This fault containment guarantees that a single fault in the FTP cannot prevent the three channels from reaching an agreement on the result of a vote. Thus, a channel or interstage may transmit bad data due to a single fault, but the bad data will be masked out by the rest of the system, which is fault free and generating correct data.

1.4 Use of the Data Exchange

A typical use for the data exchange mechanism would be a space craft control system reading a sensor. For complete fault coverage, three sensors would be used to read the same data, and each sensor would connect, through the I/O network, to a specific channel. Each channel would read a sensor and store a local value. Then, one by one, the channels would triplicate their local data by exchanging it with the other channels.

Figure 1-2 shows an example of channel A triplicating its local value via the data exchange. Note that channel A sends its local value directly to channels B and C, which route the data to their interstages. Then, all three channels initiate a vote on the raw data. The result of this vote is used by the three channels as channel A's value. This same procedure is then repeated for channels B and C. This exchange process ensures that, even in the presence of a failure, all three channels have an identical (although not necessarily correct) value for each channel's sensor reading. Thus, when this process is finished, each channel has three values that are identical to the three values that the other two channels have. The code that initiates these exchanges would be located in a library of I/O subroutines. This library is used to hide the data exchange mechanism from the user's application, preventing the user software from violating the abstraction of the FTP as a single processor. The following is an example of the code that performs a data exchange. Note that this code is executed at the same time by all three processors, giving each channel an A_value, a B_value, a C_value, and a local_value.

```
local_value := read_sensor;

A_value := exchange(from_a, local_value);

B_value := exchange(from_b, local_value);

C_value := exchange(from_c, local_value);
```

ORIGINAL PAGE IS
OF POOR QUALITY

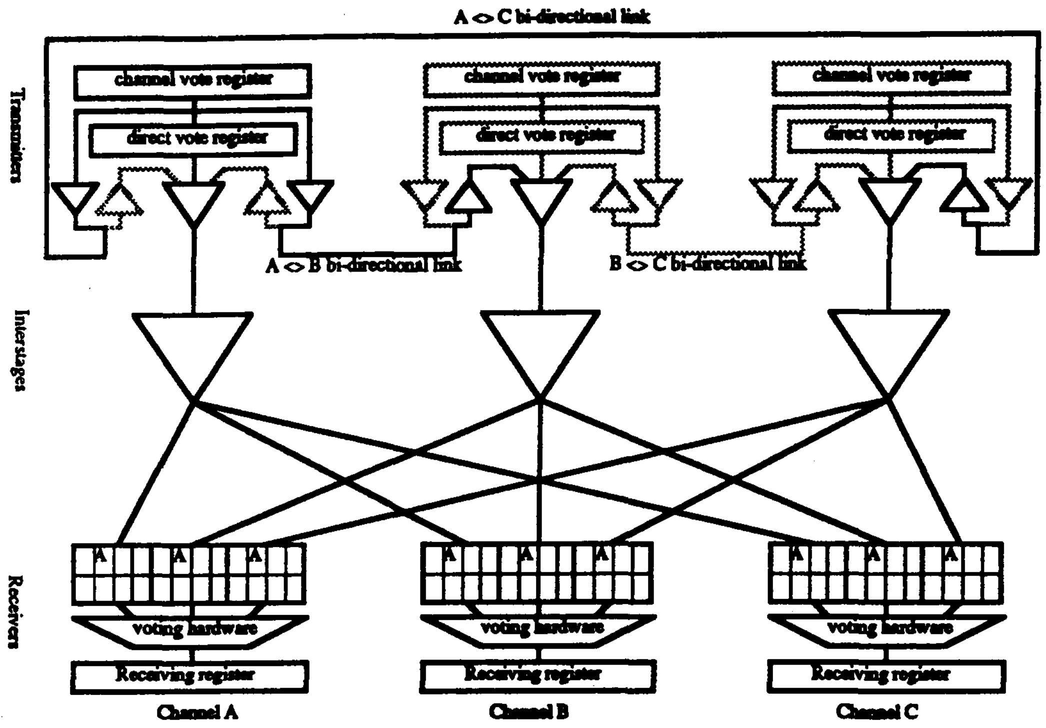


Figure 1-2: A data exchange from channel A

After all three channels have the three sensor readings, some type of redundancy algorithm (e.g., mid-value select) can be applied to these values to form a suitable result for the sensor reading. This "correct" value for the sensor reading is then used to produce an actuator command for maneuvering the space craft. Figure 1-3 shows the direct vote of this actuator command. Each channel directly sends its value to its interstage. The interstages then triplicate the data and send it to the receivers, which vote the results as before, noting any errors. Again, this whole process would be hidden from the user's application by a call to the FTP's I/O subroutine library. The output subroutine is also fairly simple:

```
voted_command := exchange(from_all, local_actuator_command);
send_command (voted_command);
```

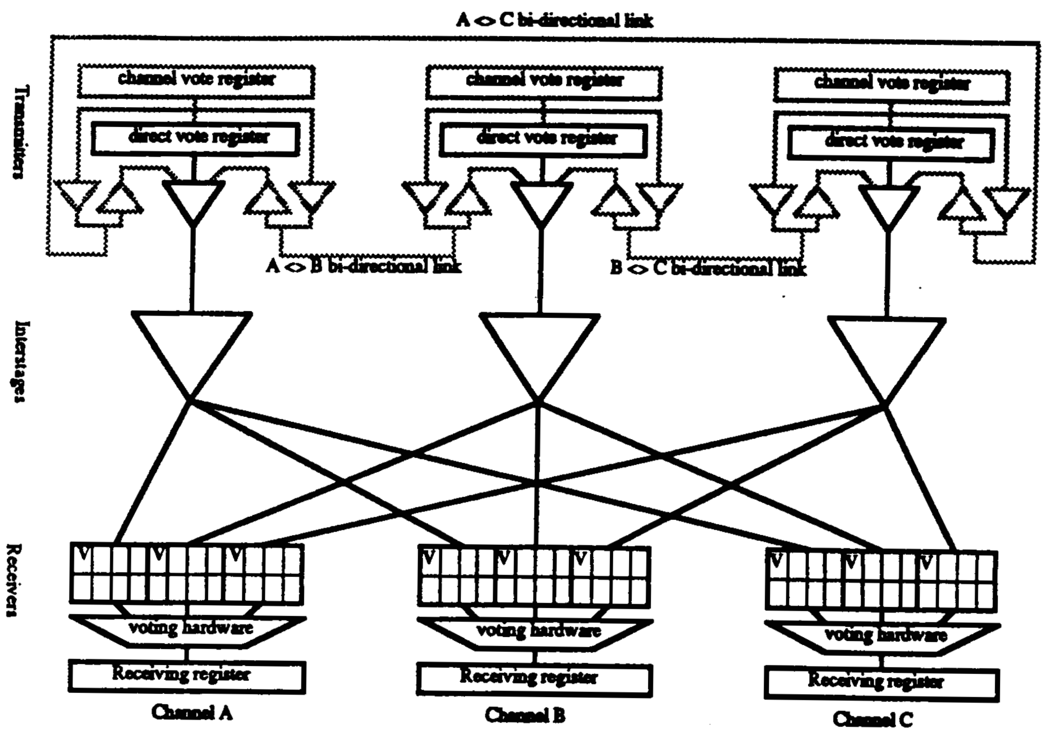


Figure 1-3: Direct vote of actuator command

ORIGINAL PAGE IS
OF POOR QUALITY

2. Structure of the FDIR code

This chapter discusses the design of the FDIR software. The design of the FDIR code was shaped by two main goals: provide complete fault coverage and use minimal processor overhead. FDIR must be able to locate and isolate any fault that occurs, and this must be done while using less that 5% of the processor's capacity. As a result of these design goals, the FDIR code is split into several tasks. The fast task can be run frequently, while the more complex tasks are run only on demand or at a lower frequency. This division allows for complete fault coverage while reducing the amount of processor time used.

In terms of software engineering, the design goals were to create FDIR code that is modular and readable. Ada helps these goals with its data abstractions and its packages, which are the advantages often cited when discussing the merits of the language [3]. Ada's use of data abstractions helps produce readable code by allowing programmers to manipulate data in a conceptual manner rather than a manner specified by the machine's representation of the data. Ada also helps produce modular code by encapsulating programs in constructs called packages which introduce these data abstractions. As a result, the FDIR code for the AIPS FTP has turned out much more modular and readable than the FDIR code that was written for a previous FTP using C.

The packages that comprise the FDIR software can be divided into four major categories:

1. Declarations
2. Resources
3. Extensions
4. Applications

2.1 Declarations

Declaration packages are collections of namings and constants that are used in many sections of the FDIR software (as well as the rest of the operating system). The only example of this type is *Memory*, the package that contains the mappings of all the special memory locations. *Memory* defines the locations for the data exchange hardware, the shared memory objects, and all the other hardware that is memory mapped, such as the timers and the Monitor Interlock.

2.2 Resources

Resource packages contain data types and operations that have general utility. For example, all the necessary procedures and types for using the data exchange and voting mechanism are defined in the *Exchange* package. Any software that is run on the FTP will need to vote input and output. The *Exchange* package encapsulates the data exchange hardware with a software abstraction so that all other software uses the voting mechanism without relying upon any implementation details. This means that if the data exchange hardware changes, only *one* package has to be changed to reflect the differences. Another resource package is the *Error_latch* package, which defines a data type for the error latches as well as the operations necessary to convert their hardware representation into a software defined Ada data type. Again, only one package relies upon the actual implementation of the hardware error latches, and only one package would have to be changed if the error latches were changed. The two remaining resource packages are *Config* and *Transient*. Both of these packages provide procedures, types, and variables that monitor the state and "health" of the FTP's three channels. The *Config* package's primary responsibility is to maintain the software record of the three channels' status: present or lost. *Transient*, on the other hand, is primarily responsible for maintaining an unreliability index for each channel. Of all these resource packages, only *Exchange* would have a system-wide utility. The other (non-FDIR) parts of the operating system, however, do require access to things such as the error latches and the current configuration of the channels.

2.3 Extensions

Extension packages are used to actually extend the Ada language. Certain operations (such as the bit wise AND of two integers) are either not permitted or difficult to implement in Ada. Extension packages, which are series of assembly language subroutines that masquerade as Ada packages, add this needed functionality to the standard language operations. The package *Memory_utilities*, for example, was created so bit wise AND and OR operations could be performed on two integers. Although Ada can actually do AND and OR operations on arrays of boolean variables, the Telesoft compiler that produces the FTP code cannot pack a 16 boolean array into a single word.

The second extension package, called *Sync_utilities*, was created for synchronizing code execution among the channels. Synchronization requires absolute control over the timing of each machine instruction. Assembly language code *had* to be used for the critical part of the synchronization procedure to meet these strict timing requirements. The *Sync_utilities* package also provides the procedure that aligns memory. The memory align could have been done in Ada, but the time penalty for not using highly optimized assembly language code to align all of memory was too great for the FTP, which is designed to be a real time system.

2.4 Applications

The FDIR application packages use the resource, declaration, and the extension packages to actually "do something." These application packages do not define any new types. Instead, they import types and low level procedures from the three other kinds of packages. In general, the FDIR application packages have only a few visible procedures, which are mostly linear code. The three application packages that make up the FDIR manager are: *FDIR*, which detects and isolates all faults; *Sync*, which synchronizes the code execution initially and whenever a channel is lost; and *Test*, which constantly runs self test on the FTP hardware.

The *FDIR* package contains the actual code for the local FDIR manager. It has only one visible procedure, *Init*, which schedules a FDIR task to be run at a relatively high frequency (approximately 16 Hz, or every 60 msec). This task, called *Fast_FDIR*, is used to spot the occurrence of errors and isolate only the most obvious faults. Both the channel presence and the interstage tests are simple enough to be run at this relatively high frequency. *Fast_FDIR* also checks all reports from other parts of the system. If there are any necessary reconfigurations, *Fast_FDIR* will do the reconfigurations in a prioritized order. This higher frequency of operation improves reliability of the FTP by reducing the amount of time an error goes undetected. In reducing this time, the window in which two errors could simultaneously occur is also reduced. A second error, if it occurred before the FTP could reconfigure around the first error, would lead to unpredictable results.

There are, however, the two competing goals for the FDIR manager: complete fault coverage, which demands high frequency, and minimal use of processor time, which demands faster, less complex operations. Thus, the *Fast_FDIR* task cannot take the time to analyze all possible fault conditions when an error is detected; it only analyzes the most simple cases. If *Fast_FDIR* encounters an error condition that it cannot analyze, then a new task is started, called *Slow_FDIR*. *Slow_FDIR* is referred to as an "on demand" task. *Fast_FDIR* will schedule it *only* if there is an error that is too complex to analyze immediately. *Slow_FDIR* will then fully analyze the error and report back to *Fast_FDIR* which channel, if any, is at fault. This split in the fault detection duties allows the FDIR manager to run quickly and often, fulfilling both goals.

There are three visible parts to the *Sync* package: *Init*, a procedure which initially synchronizes the code execution between the three channels; *Lost_soul*, a procedure which is continually run by a lone lost channel; and *Lost_soul_sync*, a task which a pair of synchronized channels will schedule (at a fairly low frequency) to find the third lost channel. These three pieces of code do exactly the same thing: send "lost soul" data patterns through the data exchange mechanism and wait for the electronic "echo" that indicates another channel was attempting to exchange at the same time. All three, in fact, use the same assembly language subroutine for the hardware interface.

The primary difference between these three operations is what they do once a channel is synchronized, when they run, and how often they run. *Init* runs only upon system initialization, and assumes that all channels are unsynchronized at the start. After two or more channels are synchronized, *Init* will reconfigure the internal FDIR records to match the new state of the hardware. Where *Init* is linear code that is run only once, *Lost_soul* is a tight loop with only one exit condition: synchronization. Any lone channel that needs to synchronize will run *Lost_soul*, and nothing else, until it resynchronizes with the other two channels. *Lost_soul* is run frequently so that whenever the other two channels find time to try to pick up the lone processor, the lone processor is waiting and ready to be picked up. The *Lost_soul_sync* task, on the other hand, is a shell that calls the *Lost_soul* procedure. The difference with the task is that the *two* channels (in synchronization) will call *Lost_soul* at a lower frequency. Also, when two or more channels execute *Lost_soul*, they only go once through the loop and exit. Thus, the *Lost_soul_sync* task can be scheduled to run at a low frequency and will only take a small amount of time to execute.

The third application package is *Test*, which contains the four FDIR self tests: voter and error latch, which verifies the voting mechanism; ROM sum, which checks the integrity of the FTP's ROM; RAM pattern, which tests the functionality of each RAM location; and RAM scrub, which ensures that all three channels have identical values in RAM. *Test*, like the *FDIR* package, has only one visible procedure, *System_test*. *System_test* calls each individual test in the appropriate order. Thus, the voters are tested before any memory values are exchanged, and the memory hardware is tested by the RAM pattern test before the memory contents are checked by the RAM scrub test. If any one of the four tests reports that there is a faulty channel, then *System_test* will stop and notify *Fast_FDIR* that a reconfiguration is required. *System_test* is called by a task in the *FDIR* package called *Selftest*. *Selftest* is scheduled to run at a low priority. Thus, if the processor has any free time, it will run some self tests.

3. The Suitability of Ada for the FTP

While developing the fault detection code for the Fault-Tolerant Processor, both the advantages and the disadvantages of using Ada were apparent. In general, the advantages of Ada, which are mostly due to the language specification, outweigh the disadvantages, which are mostly due to the compiler used for this project. This chapter discusses both the advantages and disadvantages of using Ada for the FTP, and why using Ada was, in the long run, a wise choice.

The choice of Ada as the development language was a controversial decision. Previous work on fault-tolerant processors at the Laboratory had been done in the C language, and using C would have saved the many man hours spent re-creating code that had already been written. Using C would have also meant that the software engineers would have had a familiar set of tools available to use (e.g., compilers, debuggers, etc.). But, there are two major reasons that led to the selection of Ada as the development language for the AIPS system. The first is the Department of Defense's requirement that Ada be used for military software. The second reason is Ada's tasking, exception handling, strong typing system, and enforced modularity that are widely touted in some circles [3]. The combination of these reasons led the original design team to specify that Ada would be used for the AIPS project. After almost a year of FDIR code development, the choice of Ada is still controversial.

3.1 Disadvantages

The main disadvantage of Ada is that it is an immature language. There are only a handful of fully validated compilers and few support tools for programmers. The compiler used for the AIPS FTP (the unpublished Telesoft Ada compiler version 1.5) has several specific shortcomings: the Run-Time System is inadequate for the FTP's requirements as a real-time system, the compiler produces inefficient code and is not a fully implemented version of Ada, and there are no debugging tools. Solving some of the problems associated with this system required a great deal of effort that would not have been expended if Ada were a more mature language.

The primary problem with the Telesoft Ada compiler is the Run-Time System's task scheduling mechanism. For a real-time control system such as the FTP, task scheduling is critical, and the first-in, first-out task queue supplied with the Telesoft system could not meet the strict timing requirements of a real-time system. Task priorities and interrupts are needed so that a minor task (such as a self test) would not prevent a critical task (such as *Fast_FDIR*) from running. After much work, Draper Labs developed a system of priorities and interrupts that were incorporated into the Telesoft Run-Time System. This new run-time system allows higher priority tasks to interrupt the operation of those with a lower priority and includes timing information that specifies the frequency at which a task should be scheduled. Unfortunately, the run-time system's size (approximately 48K bytes) is almost an order of magnitude larger than the operating system used for the C version of the FTP. Although the Telesoft Run-Time System code has more functionality than the C version's operating system, it is not clear that these features are needed for a real-time system. With this new run-time system, Ada's task scheduling could fulfill the FTP's requirements for real-time vehicle control.

Not only is the Telesoft Ada Run-Time System larger, but the size of the object code generated by the Telesoft Ada compiler was surprisingly large as well. In fact, the FTP system had to be redesigned to include one megabyte of RAM rather than the original 256K bytes, which would have been sufficient had this code been written in C. This increased code size has several sources: the immature compiler, which generates inefficient code, the code design, which can add to the compiler's inefficient code generation, and the required Ada runtime overhead, such as range and exception checks. Better compilers will, of course, help this problem. However, Ada rarely produces code as efficient as C, just as C rarely produces code as efficient as assembly language. Fortunately, the FDIR code has not exceeded the original C language size by any large amount, and the *Fast_FDIR* task is still within the 5% processor capacity goal.

Because the Telesoft Ada compiler is not a fully implemented version of Ada, some coding problems must be resolved in awkward ways. For example, the representation for the error latches would logically be an array or record of boolean types. The Telesoft compiler, however, does not allow the representation of an Ada record or array to be specified on a bit-by-bit basis. Thus, when the data type for an error latch was defined, Telesoft Ada could not define a record that matched the 12-bit structure and location of the actual error latches. But, because the error latches had to be exchanged among the channels as 16-bit integers, a standard record or array could not be used either. Thus, the FDIR code used a function that converts the hardware error latches into patterns that fit a 16-bit integer. Unfortunately, this sacrifices one of the primary advantages of the Ada language: its ability to easily create data abstractions from built-in types. Other problems with the Telesoft compiler were along the same vein: problems that were irritating because hardware representations could not be mapped to data abstractions with the ease that Ada promised, and solutions that were difficult to use in Ada because they did not take advantage of the built-in types and functions.

Finally, the fourth problem with the Telesoft compiler is the total lack of debugging aids. In terms of debugging tools, a disassembler is absolutely required. Thus, the Laboratory had to produce, in house, a disassembler for the FTP's 68010 code. A VAX interface program, which implements standard debugging utilities (e.g., breakpoints, memory and register displays, and program downloads), was also produced in house. Unlike the C compiler that was previously used, the Ada compiler could not produce assembly language listings of the code that have the original Ada statements inserted in the appropriate places. This was a major drawback because all matches between the disassembled object code and the original source code had to be done manually. The lack of debugging aids requires that effort be diverted from software development to debugging tool development, which is not the purpose of this project.

3.2 Advantages

On the other hand, the advantages of Ada are due largely to the language definition rather than the specific compiler. The strong typing system, for instance, allows code written by several individuals to be linked together with almost no errors. Also, Ada's package system fosters a highly modular design that clearly delineates all module dependencies, while the data abstraction capability makes it easier to create readable code. Finally, although the run-time system was not adequate at first, the Ada built-in tasking construct is useful because the FTP needs multi-processing capability.

Ada's rigid syntax and strong typing system, which are hated by some programmers, are responsible for reducing errors in software to the point that almost any program that compiles will run, and will have almost no errors. The syntax is responsible for reducing the number of typographical mistakes that are accepted by the compiler as legitimate code. The strong typing system, meanwhile, reduces the number of errors due to interfacing procedures and data abstractions. And, because the structure and syntax of Ada lets fewer errors slip past the compiler, Ada reduces the time spent debugging code.

Ada's data abstractions are a powerful force in making code that is readable and has a well defined interface. In C, for example, the configuration of the three channels (on- or off-line) was numerically represented as three bits in a 16-bit word. The representation of the data, as well as the operations performed on it, are not conceptually obvious. Ada, on the other hand, represents the configuration as a record of three booleans. Using booleans in a record to represent the configuration produces more readable code that parallels the actual structure of the information. This abstraction also reduces the mistakes and confusion between programmers who must interface code. In C, there was a convention that channel A was represented by the low order bit in a 16-bit integer. This convention, however, is not as obvious as a record with a boolean component named A. Again, Ada's data abstractions prevent these types of interfacing errors from occurring, and thereby cut the time required to debug software.

Overall, Ada is the right language for this project. The Ada language has several strong advantages, while most of the disadvantages are due to its immaturity and the specific compiler used. In time, the language will mature and more capable compilers will be available. However, even a poor version of Ada has already decreased the work required to create, debug, and interface the code on the FTP.

The decision to switch to Ada was controversial. Despite the advantages of Ada's tasking, data abstraction, and modularity, many engineers were concerned about Ada's immaturity and lack of debugging tools. Even more important, however, was the run time environment and its ability to meet the critical timing requirements of a real time control system. In spite of these problems, the development of the FDIR manager has shown that Ada has promise as a development language for embedded computer systems.

References

[1] Charles Stark Draper Laboratory.
Advanced Information Processing System (AIPS) System Specification.
Technical Report CSDL-C-5709, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, May, 1984.

[2] Alger, Linda, *et al.*
Local Fault Detection, Isolation, and Reconfiguration in a Distributed Processing System.
December, 1985.

[3] Barnes, J. G. P.
Programming in Ada.
Addison-Wesley Publishing Co., 1982.

[4] Hopkins, Albert L., Smith, T. Basil, and Lala, Jaynarayan H.
FTMP -- A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft.
Proceedings of the IEEE 66(10):1221-1239, October, 1978.

[5] Lamport, Leslie, *et al.*
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.

[6] Smith, T. Basil.
Generic Data Manipulative Primitives of Synchronous Fault-Tolerant Computer Systems.
Technical Report, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, 1980.

[7] Smith, T. Basil.
Fault-Tolerant Processor Concepts and Operation.
Technical Report, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, 1981.

N89-16340

VECTOR-MATRIX-QUATERNION, ARRAY, AND ARITHMETIC PACKAGES: ALL HAL/S FUNCTIONS IMPLEMENTED IN ADA

ABSTRACT

HAL/S avionics programmers have enjoyed a variety of tools built into a language tailored to their special requirements. Ada is designed for a broader group of applications. Rather than providing built-in tools, Ada provides the elements with which users can build their own. Standard avionics packages remain to be developed. These must enable programmers to code in Ada as they have coded in HAL/S. The packages under development at JPL will provide all of the vector-matrix, array, and arithmetic functions described in HAL/S manuals. In addition, the linear algebra package will provide all of the quaternion functions used in Shuttle steering and Galileo attitude control. Furthermore, using Ada's extensibility, many quaternion functions are being implemented as infix operations; equivalent capabilities were never implemented in HAL/S because doing so would entail modifying the compiler and expanding the language. With these packages, many HAL/S expressions will compile and execute in Ada, unchanged. Others can be converted simply by replacing the implicit HAL/S multiply operator with the Ada "*". Errors will be trapped and identified. Input/output will be convenient and readable.

Allan R. Klumpp
David D. Kwong
Mail stop 171-236
Jet Propulsion Laboratory
4800 Oak Grove Dr.
Pasadena, CA 91109
818-354-3892

To be presented at the
First International Conference on Ada Programming Language
Applications for the NASA Space Station
Hosted by the University of Houston - Clear Lake and
Johnson Space Center
Houston, Texas 77058
1986 June 2 - 5

GENERIC ADA CODE IN THE NASA SPACE STATION COMMAND, CONTROL AND COMMUNICATIONS ENVIRONMENT

D. P. McDougall
T. E. Vollman

Veda Incorporated
Lexington Park, Maryland

N89-16341

1.0 INTRODUCTION

This paper describes the results of efforts to apply powerful Ada constructs to the formatted message handling process. The goal of these efforts has been to extend the state-of-technology in message handling while at the same time producing production-quality, reusable code. The first effort was initiated in September, 1984 and delivered in April, 1985. That product, the Generic Message Handling Facility, met initial goals, has been reused, and is available in the Ada Repository on ARPANET. However, it became apparent during its development that the initial approach to building a message handler template was not optimal. As a result of this initial effort, several alternate approaches were identified, and research is now on-going to identify an improved product.

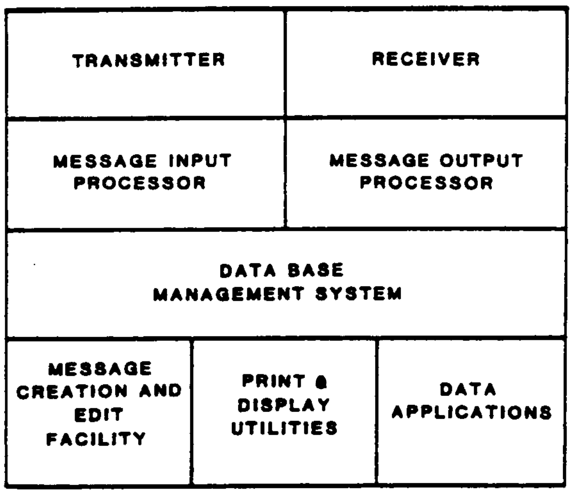
The ultimate goal is to be able to instantly build a message handling system for any message format given a specification of that message format. The problem lies in how to specify the message format, and once that is done, how to use that information to build the message handler. In Section 2 we discuss message handling systems and message types. In Section 3 we describe the "ideal" system. In Section 4 we detail the initial effort, its results and its shortcomings. We then describe the approach now being taken to build a system which will be significantly easier to implement, and once implemented, easier to use. Finally, in Section 5, we offer our conclusion.

2. BACKGROUND

Message handling systems play a major role in command, control, and communications (C3). C3 systems are most often found in military applications, where rapid, accurate dissemination of information is required. Non-military space-related communications systems face many of the same requirements. In this section, we discuss attributes of the message handling systems which support the communications aspect of C3, and we identify the requirements for those systems.

2.1 Message Handling Systems

The typical message handling systems consists of eight components, as depicted in Figure 1. The transmitter and receiver perform the actual communications between this system and some other system with which it is communicating. They handle message blocking, line protocols and other low level functions. They are usually hardware dependent and are typically written in assembler language or in microcode.


TRANSMITTER
RECEIVER
MESSAGE INPUT PROCESSOR
MESSAGE OUTPUT PROCESSOR
DATA BASE MANAGEMENT SYSTEM
MESSAGE CREATION AND EDIT FACILITY
PRINT & DISPLAY UTILITIES
DATA APPLICATIONS

MESSAGE HANDLING SYSTEM COMPONENTS
Figure 1

The message input and output processors are the interface between the rest of the system and the transmitter/receiver facility. Usually a message handling system will hold the data in some internal format which makes sense in the context of the applications to be performed upon the data. This format is usually independent of the format in which a message is transmitted or received over any specific I/O line. The message input processor accepts a bit stream input from a line by the receiver, passes it, extract the information and passes it to the Data Base Management System (DBMS). If the system provides for real-time display of incoming messages, the input processor may also pass the data along to a display utility. In a similar manner, when a message is to be transmitted, the message output processor extracts the data from the DBMS, or accepts it from a system operator, and formats a bit string (or character string) to be passed to the transmitter.

The DBMS provides for information storage and retrieval. The data may be stored in message image format, or in some non-message-related format. How the data are stored is typically dependent upon the type of applications being performed upon the data. In systems whose primary function is other than message generation and transmission, the data are not typically stored as message images. In other systems - or subsystems - whose sole or primary function is directly message related rather than data application related, the data are more likely to be stored in message image format. At any rate, when an operator creates a message, he usually wants to see its image prior to transmission; therefore the interface between the DBMS and the Message Creation and Editing facility - the editor - will normally include a utility to extract data from the DBMS and build a message in the specified format.

The editor will provide standard functions such as insert/delete line, cursor movements through the message, and so on. Additional functions to be provided are dependent upon the message format(s), which are discussed below. As we shall see, a critical function is some sort of embedded data validation.

Message handling systems usually provide the capability for visual and hard copy message output, as well as message transmission. In addition to viewing an image of the message they are creating, operators will often want to keep a hard copy of the message after it is sent, both for historical purposes, and for possible future editing.

The final component shown in the figure is not a part of the message handling system per se, but is the reason for data exchange. While there are (sub)systems whose primary purpose is nothing more than message handling (e.o. store-and-forward message drops such as the Communications Line Interface (CLI)), most message handling systems are components of larger systems which perform some applications of the data to non-transmission related problems. The data applications are not treated here; they do, however, impact the format in which the DMBS holds the message data.

Examples of message handling systems include the Force High Level Terminal (FHLT), the Ocean Surveillance Information System (OSIS), the Joint Tactical Information Distribution System (JTIDS), and the World Wide Military Command and Control System (WWMCCS) among many others. These systems employ a number of different message types, or formats.

2.2 Message Types

Examples of message types include RAINFORM (of various subtypes), Unit Reports (UNITREP), Movement Reports (MOVREP), and Joint Interoperability of Tactical Command and Control System (JINTACCS). The message formats have a number of elements in common. First, each type (or subtype) is defined to pass on data concerning a fairly specific event or of a fairly specific nature. For example a RAINFORM Green message provides tasking data to U.S. Naval forces prior to a mission, while those forces use a RAINFORM Purple message to report the results of that mission. For another example, a JINTACCS B704 is an Airbase Status Report while a JINTACCS C100 is a Imagery Interpretation Report.

Given the differing data requirements of these different message types, it would be surprising if they could all be accommodated using the same format. In fact, no such format has yet been found. However, the formats which have evolved over time have a number of similarities.

1) Messages are composed of two pieces, a header which describes the sender and the routing and other information about the message, and the body of the message holds the data content.

2) Both the header and body of the message are composed of line groupings which contain one or more lines in some specific order.

3) Each line is composed of a given sequence of fields (or components) whose appearance or order can vary only within narrow bounds.

4) Each field in a line contains a "molecule" of data which must be given in a predefined format. In fact some fields are composed of subfields (e.g. latitude is composed of degrees, minutes, cardinal point, and in some cases, checksum).

5) There are three types of fields: discrete, numeric, and text:

   a) Discrete fields are fields which must contain one of a (small) finite number of entries - for example a "month" field would have only twelve possible valid entries.

   b) Numeric fields are fields whose entries must evaluate to some numeric value. These fields may have a prescribed format as integer for fixed point. In either case, the number of significant digits (minimum and or maximum) may be specified as may the number of digits on either side of the decimal.

   c) Text fields are freeform fields whose contents may be any string of characters from some predefined character set - usually letters, digits, and some punctuation characters.

Message types differ then in which fields they use (and how each is defined), how those fields are used to define lines, and how those lines are grouped to form line groups. In addition, some message types are fixed format (the fourth field always starts at character 17) while others make use of delimiters to define where one field stops and the next starts. UNITREP is an example of a fixed format message type, while RAINFORM is an example of a "freer form" type.

The ideal message handling system would handle any and all message types with the same (or similar) sets of functions and user interface. If such a uniform system were to be built, the factors listed above define the flexibility requirements for accommodating various message type definitions.

2.3 Message Handling System Functional Requirements

Given that a message creation and editing system for some message type is to be developed, what requirements must it meet? The requirements important for the transmitter/receiver portion of a message handling system are certainly different than those which drive an editor's requirements. it appears that there are three factors which exert the most influence on an editor's requirements and design: reliability, maintainability, and date validation.

Reliability is important for two reasons. First, communications systems are usually of a critical nature, and their failure can be catastropic. Therefore, message handling systems must work predictably to ensure that the system provides the capability expected during stress periods. Second, the output of one such system is always the input of another. Therefore, the failure of a message handling system to maintain communications or to pass accurate, properly formatted data impacts the ability of other systems to meet their requirements.

Maintainability is important due to the rapidly changing nature of the communications theater. New communications systems are constantly being fielded, and existing systems being upgraded. As this occurs, new message types are being added and existing types updated. For example, one existing message type has increased in size by over 20%, in terms of the number of different line types, over the past six years. As new message types are added and existing types modified, existing message handling systems must be modified to accept these new data.

Data validation is in some sense a component of reliability, but is so critical to the mission of an editor and message handling system that we break it out separately. Newer message handling systems (and some older ones such as FHLT) provide a high degree of input message checking; messages which contain invalid data are either put into an error queue, or discarded. In the former case the valid portions of the message are only available to the system through operator intervention, in the latter case they are not available at all.

2.4 Existing System Deficiencies

The current situtation can be summarized as follows: there are a variety of message formats, each of which is handled on several message handling systems, each of which has its own custom software for each different message type it deals with. This than means that there is in fact not a single RAINFORM message handler, but several, each with its own code, its own set of functions, and its own user interface. Thus, when the RAINFORM message specification is updated, those updates find their way into some systems and not others.

This leads to the following problems:

1) Configuration management is complicated by the various implementations or message handlers for the same message types.

2) Consistency and reliability suffer due to the fact that each message handler implements somewhat different versions of the message standard in questions.

3) Maintenance is difficult and costly since each system is coded in a unique fashion, many in different languages, almost all using different approaches.

Generally speaking, each time we build a new message handler - whether for a new or existing message type - we are gaining nothing from the fact that we have ever built such a thing before. Furthermore, the costs involved in "reinventing the wheel" stay with each system throughout its lifecycle. In the case of C3 systems, the lifecycles are long and therefore the excess cost high.

Significant savings can be realized if we attempt to reduce or eliminate the scope of the problems discussed above through reusing message type definition and message editing and handling technology. This can occur in several ways, ranging from complete reuse of existing code, through partial reuse of code, to reuse of designs and message definitions. In the sections which follow, we describe some initial attempts to explore approaches to reuse of message definitions, designs, and message editor code.

3. The Ideal System

3.1 Message Format Specifications as Ada Constructs

With each message format, there exist in some form or another, a format specification. This specification provides detailed information about the message format from the level of a message as an entity on down to the field content level. This information provides guidelines required by applications programs for properly handling formatted messages. Section 2.2 above describes in some detail, the types of information provided by a format specification.

Ada lends itself very nicely to defining such specifications. A field is the lowest level defined by a format specification. As mentioned in Section 2.2, there are three basic types of fields: discrete, numeric, and text. In Ada, discrete fields may be defined as enumeration types. Numeric fields may be defined as either integer, fixed point or floating point type. Text fields may be defined as string. Compound fields may also exist. They are fields which consist of several field components, all of which must be one of the three basic field types. An example of a compound field is a latitude field. In Ada a latitude field may look like:

```
type LATITUDE_FIELD is
  record
    DEGREES        : DEGREES_FIELD;
    MINUTES        : MINUTES_FIELD;
    CARDINAL_POINT : CARDINAL_POINT_FIELD;
    CHECKSUM       : CHECKSUM_FIELD;
  end record;
```

Where the field component types: DEGREES_FIELD, MINUTES_FIELD, CARDINAL_POINT_FIELD, and CHECKSUM_FIELD have previously been defined as either discrete, numeric, or text.

In a formatted message, a line is composed of a given sequence of fields. Using Ada, a line can be represented as a record structure. Each component of the record structure would be a field. For example, a formatted line which reported the contact position of a ship may consist of three fields: contact identifier, latitude of contact, and longitude of contact. In Ada, the contact position line may look like:

```
type CONTACT_POSITION_LINE is
  record
     CONTACT_IDENTIFIER : CONTACT_IDENTIFIER_FIELD;
     CONTACT_LATITUDE   : LATITUDE_FIELD;
     CONTACT_LONGITUDE  : LONGITUDE_FIELD;
  end record;
```

Where the field types: CONTACT_IDENTIFIER_FIELD, LATITUDE_FIELD, and LONGITUDE_FIELD have previously been defined according to the rules for defining field types.

When lines are grouped together in a particular manner, they make up a formatted message. In Ada a formatted message may be represented as a record structure. Suppose a formatted message of a particular type was made up of the following five formatted lines: message identifier line, contact sighting line, contact position line, contact amplification line and a remarks line. The Ada definition of such a message type would be:

```
type FICTIOUS_MESSAGE_FORMAT is
  record
    MESSAGE_IDENTIFIER     : MESSAGE_IDENTIFIER_LINE;
    CONTACT_SIGHTING       : CONTACT_SIGHTING_LINE;
    CONTACT_POSITION       : CONTACT_POSITION_LINE;
    CONTACT_AMPLIFICATION  : CONTACT_AMPLIFICATION_LINE;
    REMARKS                : REMARKS_LINE;
  end record;
```

Where the various line types have previously been defined according to the rules for defining formatted lines.

In the "ideal" system, a message format would be defined as an Ada construct similar to that described above. Such a means for defining a message format has many advantages. In particular, the message format specification becomes a compilable unit therefore providing a means of partial validation of the format specification syntactically and semantically. Additionally, the Ada definition of the message format may be used directly in applications Ada programs that require knowledge of the format.

There are a variety of methods for defining message format specifications in Ada, however the record structure described above appears to be the most natural representation of a message format for existing formats. Currently the United States Air Force (USAF) is working with the JINTACCS community to define their message formats as Ada record types.

3.2 A Generic Message Handling System

Though message formats will vary, the requirements for message handling systems, as described in Section 2, tend to remain fairly static. Generating a message handling system for each distinct message format is a costly and time consuming task. A solution, though a non-trivial one, would be to develop in Ada, a generic message handling system. The generic message handling system would essentially be a generic package with its functions and procedures not customized around any specific message format, but rather designed to work with any message format specification that the package is instantiated for. This would imply that the only significant requirement for creating a message handling system for a particular message format would be that the specification for the message format be defined as an Ada record and then the generic would have to be instantiated for the message format. All information about the message format required by the message handling system could than be extracted from the Ada record structure containing the message format. Ideally than, the generic definition would be as follows:

```
generic
--
  type MESSAGE_FORMAT_SPECIFICATION is private;
  -- where the actual parameter here would be a record type
  -- much like that defined in Section 3.1 above
--
package MESSAGE_HANDLING_SYSTEM is
--
```

A person farmiliar with Ada generics or C3 systems would immediately identify the "ideal" system as being highly improbable. However, it is conceivable that a close approximation could be reached. The close approximation would not be as clean cut as the "ideal" but it would have many of the same benefits.

4. Striving for the Ideal System

4.1 GMHF as an Approximation

A first attempt at developing a generic message handling system was completed in April 1985. The project, Generic Message Handling Facility (GMHF), was sponsored by the USAF and the Naval Ocean Systems Center (NOSC). GMHF is not a complete message handling system. It primarily consists of the Message Creation and Editing facility. The feeling being that sufficient amounts would be learned from doing an editor and there was no real requirement to build an entire message handling system for this effort. The purpose behind this effort was three fold. First, a feasability study was to be performed to determine just how close to the "ideal" system could you get using pure Ada features. Secondly, a prototype system was to be developed as a close approximation to the "ideal" system. And thirdly, a final analysis was to be performed to determine just how cost effective the generic system was to use.

The first question was answered early on. It was apparent that there was no clean cut method for building a generic package around a generic formal parameter which was a message format specification as an Ada record like that defined in Section 3.1. Record types can indeed be passed as generic parameters, however within the generic, little can be done with the record structure since it is private.

In striving for an approximation to the "ideal" system, it became clear that some sacrifices would have to be made. Since a main concern of this effort was to determine cost effectiveness of generics in real world applications, the message format specification as an Ada record was substituted for something less sophisticated. The format specification record was replaced by several generic formal objects and types, and a database of message specification information. Additionally several procedures had to be passed as parameters to the generic. Provided below is the essence of the generic definition for GMHF. Some minor details have been left out for simplification purposes.

```
generic
--
  MAXIMUM_CHARACTERS_PER_LINE : POSITIVE;
  -- constant value telling the generic how many characters
  -- maximum a formatted line may have for the instantiated
  -- message type.
  --
  MAXIMUM_FIELDS_PER_LINE : POSITIVE;
  -- constant value telling generic how many fields maximum a
  -- a formatted line may have for the instantiated message type.
  --
  MESSAGE_FORMAT_FILE_NAME : STRING;
  -- constant value providing the name of the file which contains
  -- the message format specification
  --
  type LIST_OF_FIELD_NAMES is (<>);
  -- an enumerated list of all fields for the message type
  --
  type LIST_OF_LINE_NAMES is (<>);
  -- an enumerated list of all lines for the message type.
  -- the line names are keys into the message format
  -- specification file.
  --
  with procedure GET_FIELD( FIELD_NAME  : in  LIST_OF_FIELD_NAMES;
                            FIELD_VALUE : out STRING ) is   ;
  -- this procedure provides all instantiations of I/O packages
  -- for the field data types of a message type. In addition,
  -- the routine is organized as a large case statement which
  -- calls the appropriate input routine for a given field type
  -- upon request. This has proved to be a long and tedious
  -- routine to generate.
  --
  with procedure FORMAT_LINE_OF_TEXT(LINE_OF_TEXT: in out STRING)
                                      is NULL_PROCEDURE;
  -- this procedure handles the formatting of a line of text so as
  -- its physical appearence meets the requirements of the
  -- specification.  For example, JINTACCS requires a '/' as a
  -- field delimitter between fields. When a field is left blank
  -- it appears as a '/-/'in the text of the message. This
  -- procedure would be responsible for identifing a field as
  -- being blank and subsequentlly placing a hyphen in the text.
--
package MESSAGE_HANDLING_SYSTEM is
--
```

The new types and objects as formal parameters and the format specification database contain much of the same information as the record construct would have, but with great redundancies and in a less clean, less natural fashion. The end result of all this was a generic message handler which was a successful system but not an optimal one.

With the successful development of a generic message handler, the question of cost effectiveness still remained. To resolve this question, the generic was instantiated for two message types, RAINFORM and UNITREP. The RAINFORM instantiation was completed by one of the developers of the generic software. The UNITREP instantiation was completed by an individual only mildly familiar with the software but very farmiliar with Ada, the idea being that the average instantiator of the generic would know little or nothing about the software itself. The results were very promising. RAINFORM required a fairly significant amount of time for instantiation, about 300 man hours. The majority of this time was spent debugging problems in the generic which were encountered for the first time. The instantiation of the UNITREP message handler took approximately 60 man hours. The time for producing the UNITREP instance was significantly less then the time that would have been required to develop a non generic message handling system unique to the UNITREP message format.

In short, development of GMHF and instantion for RAINFORM and UNITREP message formats yielded one set of positive results. Use of generics in real world applications should prove to be a very cost effective means of software development. At the conclusion of the GMHF effort, the question was raised, " Are there alternative means for developing message handling systems which are better than those imposed by GMHF? "

4.2 Problems with GMHF

To determine better means for developing message handling systems, an attempt was made to identify problems and deficiencies with GMHF. One deficiency was immediately apparent. GMHF required that the use of message format specifications as records be sacrificed, so that we could develop the system as a generic unit. In place of the record structure, an implementer of the generic was forced to define data types to pass as formal parameters which would normally not have been required. In addition, a small database of message format specifications had to be created by the implementer for use by the generic. These undesirable work-arounds preferably should be avoided in future systems.

A requirement of message handling systems is that they support data input and output (I/O) operations, data validation, etc. The DBMS and Message Creation and Editing facility discussed in Section 2.2 above, clearly have this requirement. I/O operations in this case do not refer to the low level I/O required by the transmitter and receiver, but rather to the I/O routines obtainable by instantiating packages such as TEXT_IO, INTEGER_IO, ENUMERATION_IO, DIRECT_IO, etc.. GMHF supports the I/O requirements, but with one hook. All I/O functions and procedures which are to operate on types defined outside of the generic must be themselves defined outside of the generic and passed into the generic as parameters. This seems like an obvious requirement and it is. Obvious as it may be, it is a tedious, therefore undesirable task instantiating I/O packages for the types and subtypes which comprise the many fields of a message format oftentimes numbering in the hundreds.

To summarize, if the amount of work required by the implementer of the message handling system could be reduced to a minimum, such a system would become a much more powerful, useful tool. Therefore we must solve two problems. First, a way to utilize the record definition of a message format specification must be developed. Secondly, the requirement for the implementer to provide instantiations of all I/O packages for the different field types and subtypes must be eliminated, vastly improving the usability of the system.

Through careful investigation it became clear, there is no clean cut solution. Either you part with the message format specifications as records, or you must part with the idea of a generic message handling system. And in either case, the I/O packages for each of the field types would have to be created or instantiated by the implementer of the system.

4.3 Introducing a Preprocessor to the Problem

Following the conclusive results of GMHF, a new concept was introduced. A preprocessor could be developed which would accept as input the message format specification as a record type, and output as Ada code, a compilable package specification containing all types, instantiations of I/O packages, etc., required to instantiate the generic message handling system. Essentially, this allows us to obtain the desired goal. An implementer is only required to generate a message format specification as an Ada record and then instantiate the generic. Of course there are some rules to follow when defining the message format specification so as to stay within the bounds of the preprocessor. The development of such a system is currently in progress with an expected completion time frame of September 1986. Portions of the system are being developed under contract to the USAF and NOSC, while the basis of the preprocessor has already been developed by a third party as an internal research and development project.

4.4 Implementation of Such A System

Implementation of such a system can be described as a series of three main steps.

4.4.1 The Message Format Specification

The implementer is first required to generate a package specification containing the record type definition for the message format as demonstrated in Section 3.1 above. Having completed this, the package specification should be compiled to validate it syntactically and semantically.

4.4.2 The Preprocessor

Having successfully compiled the message format package specification, the preprocessor should be activated. The preprocessor will read the message format package specification as an input file and generate an output file which is also a package specification. The output file will contain all types, I/O packages, etc. derivable from the input package specification which are required for instantiation of the generic message handling system.

4.4.3 The Output Package Specification

When the preprocessor is complete, the output package specification should be compiled. The implementers applications program may then "with" the compiled output package specification and in turn, instantiate the generic message handling system. There will be additional generic parameters which the implementer will be required to provide for the instantiation which will not be included as part of the package specification output by the preprocessor.

5. Conclusion

5.1 Status

The preprocessor solution for the "ideal" system is midway through the design phase. Currently a prototype of the message handling system is being developed to determine specific requirements for the output of the preprocessor. A preliminary version of the preprocessor has been developed, however not with this particular application in mind. An expected completion date for the entire system is September, 1986. The system will be made available in the public domain via ARPANET upon completion.

5.2 Summary

Development of a system such as that proposed by the preprocessor method could in a sense, revolutionize the use of message handling systems in the C3 world. Currently, so much money is poured into the development and maintenance of systems in support of C3. To begin development of code for such systems in Ada is a very large step to improve the reliability, maintainability, and reuseability of such systems. Additionally, the generic message handling system as described in this paper would be a welcome asset to the development of C3 systems. The generic message handling system is portable between hardware, and implementable for most every message format in use today by the DoD.

N89-16342

IMPLEMENTING DISTRIBUTED Ada[1] FOR REAL-TIME APPLICATIONS

Patrick Rogers
Charles W. McKay
High Technologies Laboratory
University of Houston
at Clear Lake

Abstract

The discussion of applying a (distributed) High Order Language (HOL) to applications which require real-time performance invariably invokes the subject of excessive overhead. In a related paper,[2] the authors discuss some of the basic language-specific issues involved in distributing a High Order Language, with special attention paid to the Ada language. In a traditional implementation, several of these issues imply considerable, if not prohibitive, overhead. In this paper, an implementation strategy is introduced which promises to deal with these issues in a manner that will provide significant performance improvements. These improvements should in fact be sufficient to make use of distributed Ada feasible even in highly-constrained application domains. Additionally, the general approach should be applicable to non-distributed implementations as well.

[1] Ada is a registered trademark of the U.S. Government (AJPO)

[2] Rogers, Patrick and Dr. Charles W. McKay, Distributing Program Entities In Ada, University of Houston at Clear Lake (High Technologies Laboratory), Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, June, 1986

N89-16343

RT_BUILD : An Expert Programmer for Implementing and Simulating Ada Real-Time Control Software

Larry L. Lehman[*]
Steve Houtchens[**]
Massoud Navab[**]
Sunil C. Shah[†]
Integrated Systems Inc.
101 University Ave.
Palo Alto, CA 94301
(415)-853-8400

Abstract

RT_BUILD™ is an expert control system programmer that creates real-time Ada™ code from block-diagram descriptions of control systems. Since RT_BUILD embodies substantial knowledge about the implementation of real-time control systems, it can perform many, if not most of the functions normally performed by human real-time programmers. Though much basic research has been done in automatic programming, RT_BUILD appears to be the first application of this research to an important problem in flight control system development. In particular, RT_BUILD was designed to directly increase productivity and reliability for control implementations of large complex systems.

RT_BUILD Capabilities

RT_BUILD implements control systems designed with the MATRIX_X™ control design package. Control systems are specified as nonlinear, multi-rate, discrete-time block-diagrams in the interactive graphical environment of MATRIX_X's SYSTEM_BUILD™ module. RT_BUILD accesses the SYSTEM_BUILD database to create an exact implementation of the design.

The code produced by RT_BUILD includes all specified dynamic compensation and control logic, a real-time application level executive, and generic software interfaces to hardware such as sensors, actuators, and displays.

* *Manager, Software Development*
** *Member, Technical Staff*
† *Manager, Real-Time Systems*

The efficient executive performs the task scheduling and interrupt handling required to implement real-time multi-rate controllers. Initialization/termination and exception handling functions are also included.

The software elements for an application are automatically assembled from a modular library of Ada control functions. User supplied control or interface functions (such as specific hardware drivers or existing Ada control algorithms) can also be included from user libraries.

The real-time code produced by RT_BUILD is highly optimized for speed. Further optimization (speed or memory) is normally performed by optimizing compilers. The net result is an automatic implementation of the design whose real-time performance is extremely difficult to surpass.

RT_BUILD can be used at the final stage of the control engineering cycle to generate real-time code for most control problems including: aircraft/spacecraft control, robotics, process control, servo control, and any other real-time Ada application. Earlier in the engineering cycle, RT_BUILD can be used to develop off-line simulations or real-time hardware-in-the-loop simulations. Since RT_BUILD can be used to rapidly estimate real-time processing requirements, implementation considerations and processor constraints can be considered much earlier in the control development cycle.

RT_BUILD Design Goals

RT_BUILD was designed to address the requirements of flight software. The following aspects distinguish flight software from real-time software used in prototype testing.

(i) Flight software is used over a long time period. It must be very easy to maintain, update, and verify.

(ii) Processing power and memory are at a premium. They must be used efficiently.

(iii) The real-time software must be capable of handling a variety of input-output mediums.

(iv) The software must handle multi-processor implementations, since most flight control systems must use many processors.

(v) The software must accurately implement the designed control systems since costs of making errors can be very high.

The paper will describe how recent developments in computer-aided engineering have been applied to meet these requirements in RT_BUILD.

RT_BUILD Design Approach

Real-time software is generated around the concept of a modular, reconfigurable periodic scheduler and associated computation modules that can coexist with other foreground computing tasks (e.g., interrupt service routines) as well lower priority background processes. Device independent interfaces are incorporated in this design to isolate hardware dependencies. The Ada Periodic Scheduler (APS) is designed to provide periodic pre-emptive priority-based execution of tasks. The scheduler as well as the rest of the software is written in Ada.

The detailed paper will show various details of the design procedures and software structure as well as the role Ada capabilities play in this automatic code generation.

A Multi-Disciplinary Technology

The paper will discuss the multi-disciplinary technology required to develop an integrated set of Computer-Aided Control Engineering (CACE) tools which include automatic code generation capabilities.

While current tools provide a tremendous improvement in flight control systems development, areas where further research is being conducted will be covered in the detailed paper and the presentation.

TM RT_BUILD, MATRIX$_X$, and SYSTEM_BUILD are trademarks of Integrated Systems Inc., Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office).

N89-16344

A MULTICOMPUTER AND REAL TIME ADA ENVIRONMENT

Ray Naeini

Flexible Computer Corporation
1801 Royal Lane
Bldg. 8
Dallas, Tx. 75229
214/869-1234

A MULTICOMPUTER AND REAL TIME ADA ENVIRONMENT

A multicomputer is defined as a set of tightly-coupled yet autonomous computers capable of synchronizing and communicating in parallel but also of operating independently. This paper attempts to discuss the architectural concepts and requirements for executing the Ada programs in a multicomputer system. Synchronization, communication and protection of shared data between Ada program entities are addressed. Decomposition or partitioning of the Ada programs in a multicomputer system is also studied.

Finally, a multicomputer and real time Ada environment is described using FLEX/32 multicomputer system.

N89-16345

RUN-TIME IMPLEMENTATION ISSUES FOR REAL-TIME EMBEDDED ADA*

RUTH A. MAULE

Software Technology
Boeing Aerospace Company
P. O. Box 3999, M/S 82-53
Seattle, Washington 98124
Telephone: (206) 773-8607

ABSTRACT

A motivating factor in the development of Ada as the department of defense standard language was the high cost of embedded system software development. It was with embedded system requirements in mind that many of the features of the language were incorporated. Yet it is the designers of embedded systems that seem to comprise the majority of the Ada community dissatisfied with the language. There are a variety of reasons for this dissatisfaction, but many seem to be related in some way to the Ada run-time support system.

One of the more common complaints about the run-time system is that it is too big or too slow, that Ada requires excessive or unnecessary control structures to support high level language constructs that may not be used by an application. Another commom complaint is that the tasking model does not provide the type of real-time control designers are accustomed to, that the delay statement is flawed, and that rendezvous' are too expensive. These are fairly general complaints, and may well reflect real problems with the language. But there is a more fundamental problem with Ada run-time support systems of which many people are not yet aware, and that is the large number of implementation dependent characteristics which present portability problems and performance inconsistencies among validated compilers. The Ada run-time support system represents not merely a large block of additional code that must be loaded with each application, but an interface to the hardware from the source code, a real-time executive, a memory manager, and a tasking supervisor and

*Ada is a registered trademark of the United States Government (Ada Joint Program Office).

scheduler. As long as the more general Ada semantics are supported, the implementation of each of these is left largely up to the implementer. No standard interface exists between the run-time system and the application code, no consistent terminology is available for comparisons between different vendors, and no standard format defines the Ada Language Reference Manual (LRM) Appendix F, the only place where a vendor is required to describe the implementation dependent features of the system.

This paper presents some of the areas in which these inconsistencies have been found to have the greatest impact on performance from the standpoint of real-time systems. In particular, a large part of the duties of the tasking supervisor are subject to the design decisions of the implementer. These include scheduling, rendezvous, delay processing, and task activation and termination. Some of the more general issues presented include time and space efficiencies, generic expansions, memory management, pragmas, and tracing features. As validated compilers become available for bare computer targets, it is important for a designer to be aware that, at least for many real-time issues, all validated Ada compilers are not created equal.

1.0 INTRODUCTION

The high cost of mission critical embedded software was a major factor in the decision by the Department of Defense (DoD) to standardize on a single high level language (HOL). Major design decisions for this language, subsequently named Ada, were driven by the needs of embedded systems applications. Yet it is the designers of such applications that are currently among the most dissatisfied users of Ada. For many, the use of Ada is being treated with, at best, reluctant acceptance, and at worst, outright refusal.

Unfortunately, the reasons for this reaction are clear. A single language able to support the broad spectrum of DoD embedded applications must be comprehensive. Translate that to "complex," and read that "big" [Hc80], [Wb84]. For computer software, big nearly always implies "slow," and big and slow are not desirable adjectives for real-time embedded systems.

Ada is indeed comprehensive. Providing parallel processing, exception handling, and machine dependent facilities as well as structured programming support capabilities such as strong typing, modularity, readability, and generic definitions, Ada seems to have something for everyone. And early implementations have, as expected, proven to be less efficient in terms of timing and sizing than those of previous HOL's.

Then there is the policy of Ada standardization via formal validation. Where previously there was close interaction between compiler implementers and systems designers in order to develop project-specific run-time protocols and interfaces, now Ada compiler implementers must make virtually "sterile" design decisions based on the mandated necessity of passing the Ada Compiler Validation Capability test suite.

Finally, the acceptance of Ada requires the acceptance of a special, extra execution support package, the Ada run-time support system. This run-time support is required above and beyond that provided by the computer's operating executive and the application code to support Ada semantics. It appears as object code at execution time, providing many of the support functions previously designed and written by the embedded applications designer. Yet it is basically "canned" by a supplier who is unfamiliar with any project-specific needs.

The following sections are an initial view of Ada run-time support systems issues which must be defined and understood in order to make effective use of this new HOL for embedded real-time applications.

2.0 REAL-TIME ISSUES

It is to be expected that the most common complaints heard from real-time programmers about the Ada run-time support system is that it is too big and too slow. In addition, there is dissatisfaction with the tasking model, the delay statement, rendezvous costs, exception handling overhead, context switches, and more. Real-time designers, accustomed to using assembly language or HOL's not requiring additional run-time support, find it difficult to deal with the overhead that accompanies an Ada program.

A major stumbling block is presented by the Ada tasking model. Although a necessary and correctly included feature of the language, it seems that not quite enough home-work was done by the language designers to support the needs of real-time applications. Traditional approaches to real-time operating systems have relied on precise timing and tight control over the sequence and length of execution of individual system components. The Ada tasking model does not support this type of control. As pointed out by MacLaren [Ml80], the cyclic executive approach, which is most commonly applied in real-time situations, is based on synchronous sequential execution, while Ada tasking is by definition asynchronous and concurrent.

Resolutions of this conflict fall into three basic categories. Some real-time designers simply refuse to use Ada tasking, and write the executive as they would have in a language that does not support tasking. The second approach is to try to force the

Ada model to fit the solution. Finally, the solution can be redesigned, making full use of the approrriate Ada constructs. Each approach will succeed in certain situations, but the third approach is the one that must be chosen if Ada is to succeed in the long run. It is also the one least likely to be chosen, as experienced designers will be reluctant to abandon the "tried and true" cyclic executive.

3.0 AREAS OF CONCERN

The intent of this paper is to point out the areas of the Ada run-time system where implementation choices can affect real-time performance. The elements affecting run-time performance are so broad that the scope of this paper will have to be limited to some reasonable subset of them.

It is first assumed that the run-time system is written for a bare target. In this situation, the run-time system is fully responsible for run-time efficiency. It is also assumed that the target is a uniprocessor. The problems of multi-processing/multi-programming systems will be left to the more ambitious. Finally, no distinction is made as to whether the constructs discussed are actually in the run-time library or generated by the code generator. As noted before, there is no standard for this interface.

The main areas of concern seem to fall into some rather general categories, but defy rigid classification. To bring order to the discussion, the areas will be loosely grouped under the following categories: capabilities, control, kernel support, and tasking support. Capabilities and control can be looked at on either an individual level or a system level. The kernel support basically refers to the problems and elements within a program without tasking. The tasking support includes all the elements necessary to implement Ada tasking. Under kernel support and tasking support, individual system elements will be explored.

3.1 CAPABILITIES

Run-time support capabilities are those features of the run-time system that affect the application system's ability to perform its function. They represent a measure of the limitations and performance of the system with respect to what it can do and how well it is able to do it. What it can do refers to such features as support for or actual inclusion of hardware drivers, extended memory capacity, or the degree of accuracy supported for fixed or floating point calculations. How well it can do it refers to performance features such as space and time efficiency of the run-time system itself, and also of the compiler generated code. An obvious example of this is the level of code optimization achieved by the compiler. A more subtle example is the modularity of the run-time system and the user's ability to select and load only those features that are used in an application.

Specific examples of capabilities that may be necessary are numerous. The ability to specify an absolute load location for a section of code. A tracing feature that tracks the scheduler of the tasking system in real-time may be the only way to recover after a crash or to track down an elusive bug. It may be necessary to have a certain degree of accuracy for fixed point calculations. For the MIL-STD-1750A Instruction Set Architecture, the extended memory option is not a trivial addition to a run-time system.

3.2 CONTROL

Control here refers to the amount of influence possible over the system. One method of exerting control is through pragmas. Most systems for embedded targets have not fully implemented even the standard pragmas of the LRM. This will change, however, and more and more additional pragmas will be implemented as well. In this way, the vendor will be able to include special functions or capabilities not required by the standard, but of real value to a user. For example, pragmas will be able to specify that some function is not required by the application, and allow the system to eliminate that function from the resulting object code, saving at least space and quite possibly execution time.

Pragmas can also be used to request that the system do things in certain ways. Pragma optimize requests that optimization be done with one of those two often opposing goals, space and time, as the main objective. Pragma time-slice requests the system to implement a time-slicing algorithm among tasks of equal priorities. Pragma inline requests that a subprogram be included inline at each call. A pragma could be included to specify whether generic expansions should be done similarly to assembly language macro expansion, creating unique instances of code for each instantiation or if code should be shared as much as possible. Allowing the programmer to specify this information in pragmas lets the system take advantage of application-specific knowledge.

For critical real-time applications, it is useful to be able to include only those functions needed in the run-time system code that is loaded with the application. If the system is well structured, excess code could be cut from the run-time system at the source level, creating a subset system to be compiled and linked to the critical applications. One good example of this is removal of the entire tasking system. The legal aspects of this with respect to validations, however, are unknown. Another alternative is to use some type of smart linker or pre-linker that will only link in the portions of the run-time system that are necessary. This requires cooperation from the compiler, as often identification of the required run-time support routines must be done at compile time. Although the end result of these two approaches is the same, the latter one seems to be more "legal" than the former.

3.3 KERNEL SUPPORT

The implementation of the run-time kernel is of primary concern to real-time system designers. The basic support required from an Ada run-time system includes elements such as exception and interrupt handling, system clock functions, system initialization, and memory management. The effective use of registers, storage for stacks and heaps, implementation of activation records and scoping, and parameter passing mechanisms will all affect system performance.

3.3.1 SYSTEM CLOCK

The implementation of the system clock can be an important factor in the overall capabilities of the system. The counter timer chip used to drive the system clock defines the minimum granularity of time available to the system. The second level of granularity is the basic clock period, which can be found in the Ada package SYSTEM (SYSTEM.TICK) [US83]. A different level of granularity is represented by the Ada type "duration", which is not required to be the same as SYSTEM.TICK. The relationship between these values impact the system in different ways.

There is not usually a practical use for the finest granularity available from the hardware. Typically, some reasonable value is chosen for the size of the clock period, and an interrupt is generated at this rate. The interrupt handler updates the system clock, and this represents the finest resolution available. Note that, if the main processor is responsible for clock maintenance, as the resolution increases, so does the amount of time spent handling interrupts and maintaining the clock. (This is not the case if the clock is maintained independently of the CPU.)

The Ada type "duration" is not required to have the same resolution as the clock period. It is required by the Ada LRM to be at most 20 milliseconds, and is recommended that it be no more than 50 microseconds. A real-time system has timing constraints that require response within given time intervals. The clock period or the resolution of type duration must support these requirements. One system studied was found to accept a higher resolution of type duration than the system clock would permit. Although this may seem wrong at first, it was possible on that system to determine the displacement into the current clock period, and a separate timer was available for purposes of releasing the delayed tasks at the finer resolution. This is potentially more efficient, allowing finer resolution to be maintained only when necessary.

3.4 TASKING SUPPORT

The tasking supervisor typically comprises a major portion of the run-time system. In this area, many variations in implementation can appear, and can have great impact on the run-time performance of any program that includes tasking. The Ada LRM has defined the interface to the tasking system from an applications program, and a method of communication and synchronization between tasks, but has left a large part of the implementation of that system undefined. The implementer is constantly faced with a choice bewtween doing something "bare bones" quick, efficient, and simple, as would be necessary only to satisfy the LRM requirements, or going further and including features that, although not required by the LRM, are known to be highly desirable for real-time processing. If the decision is made to go beyond the requirements, then the question becomes how far to go. Added complexity will adversely affect performance, and it can be difficult to determine what is acceptable and what is not.

3.4.1 SCHEDULING

Task scheduling is an important consideration for a multitasking application. The Ada LRM does not specify a scheduling algorithm for tasks of equal priorities. Even for tasks of differing priorities, the requirementrs indicate that the task with the higher priority should be running, but the wording still leaves room for argument. Also of concern here are queueing structures, priorities, pre-emptive priority scheduling, and time slicing.

3.4.1.1 QUEUEING STRUCTURES

The order in which tasks (of equal priority) are initially put on the ready (or run) queue should be of little consequence. The Ada LRM states that programs that depend on the order of scheduling are incorrect. What is important is the method used at run time to reschedule tasks as they become ready again after a delay or some other blocked state. In theory, higher priority tasks should be dispatched before those with lower priorities. Within each priority, some fair method of sharing the processor should be implemented to prevent starvation of any single task. This may seem obvious, but the Ada LRM does not specify prevention of starvation. In fact, as long as a task does not block itself, and in the absence of synchronization through rendezvous, it is legal to allow each task to execute to completion before beginning execution of the next task. And unless some method of pre-emption is used, even a task of higher priority that becomes ready while a task of lower priority is running is allowed to wait until the currently running task relinquishes the processor.

The implementation of the queuing structures is not generally a factor in performance, but the ordering and maintenance of the queues is. The run-time system minimally provides a delay queue and a ready queue, and may additionally have a ready queue for each priority or may simply order the ready queue by priority. There may be many other queues in the system on which a task may be placed, but, (with the exception of rendezvous entry queues which will be discussed in another section) these should not affect scheduling order. The ready queue(s) should be ordered first by task priority. Within each priority, the queues should ideally be FIFO, but this is at the discretion of the implementer. The delay queue is optimally ordered by wake up time, the next task ready to wake up being at the front of the queue.

3.4.1.2 PRIORITIES

Priorities are supposed to be static except during a rendezvous, and if they are, then their effect on scheduling should be straight-forward. Some vendors may be developing some method of implementing dynamic priorities, and this will require dynamic modification of the ready queue, but as of this time, none have been announced.

Another issue in regards to priorities is whether or not higher priority tasks that become ready can pre-empt a currently running task of lower priority. This is a critical issue for many real time applications. The most common instance in which this becomes necessary is when a high priority task has been delayed for a given time span and that time span expires. The high priority task should then be allowed to pre-empt the processor from any lower priority task that may be running at the time. The alternative to this is to make the high priority task wait until the lower priority task relinquishes the processor, and then allow it to take precedence over all other ready tasks of lower priority. This is intolerable for real-time applications.

3.4.1.3 TIME SLICING

A final issue on scheduling is time slicing. Although overhead is required to implement time slicing, it is a good way to insure that each task within a priority will get an even chance at processing time. Some implementations may allow any task to be assigned an independent length of time for its time slice, or a single value may be available for modification to specify the slice length for all tasks. The user may be able to turn time slicing on and off through a software toggle. If time slicing has been implemented in conjunction with pre-emptive priority scheduling, the algorithm must take into consideration the time remaining in the slice allotted to a task that gets pre-empted so that it will be allowed to finish its slice when scheduling returns to that priority level.

3.4.2 CONTEXT SWITCHING

A terminology that is popular to toss around is the time required for a context switch between tasks. The code required and the time it takes to execute the actual context switch (that is, to change the registers, stack pointer, program counter, etc) from one task to the next are extremely hardware dependent. It is not apparent that this time has any relation to the time it takes to invoke a different task, since there is a tremendous amount of overhead involved in supporting Ada tasking that must also be done at that time. The real question is how long it takes to get a different task running once the first has given up the processor, and this reflects the amount of overhead inherent in the tasking system itself.

3.4.3 TASK TERMINATION

The part of the run-time system devoted to managing task termination can be quite extensive. The tedious bookkeeping of dependence on masters, and status of children and sibling tasks is necessary to insure that tasks terminate properly. In many cases, this overhead is necessary, but in some situations, an application may want to do away with this overhead if it does not have tasks that terminate. \This is another case where the structure of the run-time system will determine the possibility of removing or disabling this part of the system.

3.4.4 DELAY PROCESSING

When a task executes a delay statement, the run-time system must calculate the wake-up time, update the delay queue, possibly set or reset the wake-up timer, and dispatch a new task. Depending on the implementation, other functions may be required. These should be done in as efficient a manner as possible.

The processing of delay expirations can be handled in a variety of ways. If pre-emptive priority scheduling has been implemented, then a delay expiration may become a scheduling event, as the task whose delay has expired may be of a higher priority than the currently running task. To implement this, a separate timer may be set for the next scheduled delay expiration, and the code to release the task to the ready queue may be included within the handler for this interrupt. Or the code may be included within the dispatcher, and the interrupt may return to the Ada application through the dispatcher. This method forces the system to run through the scheduling routine, which may not be necessary if the released task is not of a higher priority than the running task. The code may also be included in the handler for the clock interrupt, if the resolution of type duration is equivalent to the clock period. In any case, pre-emptive scheduling requires the use of some method of regaining control of the processor by the run-time system. The efficiency with which this is accomplished is the only real consideration.

If scheduling is not pre-emptive, then the processing of a delay expiration can simply wait until the next scheduling event, whenever that occurs. Wake-up timers are not involved, and less overhead is required. But for real-time systems, this method is not really an option.

3.4.5 RENDEZVOUS COSTS

Rendezvous are effectively similar to procedure calls, yet they are much more complex to implement, and therefore create a tremendous amount of overhead for the run-time system. One task must always wait for the other to reach the point of the rendezvous, the system must invoke the rendezvous when both tasks are ready, and context switches are required between the tasks during the rendezvous. Priorities are not static during a rendezvous, and this presents additional overhead.

It is possible for the run-time system to optimize rendezvous so that the cost is more on par with that of a procedure call, but some preparation must be done at compile time. No context switch should be made if none is necessary, (such as when no code is associated with the accept statement) or when the code does not require one. This can be difficult to determine, and must be done at compile time.

4.0 CONCLUSIONS

As compilers targeted to bare computers become more common, the range of quality in run-time performance will become more apparent. Currently, many embedded systems designers are unaware of the variances simply because Ada is so new. These designers have long known what to look for in a good compiler, but many have no idea what to ask a vendor about the run-time support system.

The best run-time support for any application is determined by the individual needs of the application itself. But in a general sense, a good Ada implementation meets the basic LRM requirements, is of high and consistent quality, and is adaptable to the needs of the user. It is hoped that this paper and its successors will assist in defining the issues which impact the design and implementation of Ada run-time support systems. Once this is accomplished, ease of understanding and use should become more readily possible, allowing Ada to satisfy its requirements and original intent.

BIBLIOGRAPHY

[AR85] ARTEWG, *"Draft Charter for the Ada Runtime Environment Working Group"*, July 17, 1985.

[BR84] Baker, T.P. and G.A. Riccardi, *"Ada Tasking: From Semantics to Efficient Implementation"*, Florida State University, November, 1984.

[BS85] Baker, T.P. and G. Scallon, *"An Architecture for Real-Time Software Systems"*, Boeing Aerospace Company, Seattle, WA, July, 1985.

[GL83] Gligor, V.D. and G.L. Luchenbaugh, *"An Assessment of the Real-Time Requirements for Programming Environments and Languages"*, IEEE, 1983.

[Hc80] Hoare, C.A.R., *"The Emperor's Old Clothes"*, Communications of the ACM, Vol 24, No. 2, Feb., 1981.

[Ml80] MacLaren, L., *"Evolving Toward Ada in Real Time Systems"*, Boeing Military Airplane Company, Seattle, WA, 1980.

[US83] United States Department of Defense, *"Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A"*, Feb., 1983.

[Wb84] Wichmann, B.A., *"Is Ada Too Big? A Designer Answers the Critics"*, Communications of the ACM, Vol 27, No. 2, pp. 98-103, Feb. 1984

BIOGRAPHY

Ruth A. Maule is a systems analyst for the Ada language group of the Software Technology Division of Boeing Aerospace Company. She is currently doing research in the area of Ada run-time issues with respect to real-time embedded systems, and has written an internal Boeing document on the evaluation of Ada ru -time environments. She has done modifications to run-time systems currently in use at Boeing. She is a principal member of the SIGAda Ada Run-Time Environment Working Group (ARTEWG). She received MSCS and BSCS degrees from Florida State University in 1985 and 1983 respectively.

INTERESTING VIEWPOINTS TO THOSE WHO WILL PUT Ada* INTO PRACTICE

Arne Carlsson
Saab Space AB
Goteborg, Sweden

INTRODUCTION

Ada will most probably be used as programming language for computers in the NASA Space Station project. There will be a great number of computers and computer types, e.g. in space for Data Management System, Crew Working Station, experiments and on ground for flight control, launch control, maintenace, validation, integration, software development. Will Ada be used for all these computers or only for some of them? It is reasonable to suppose that Ada will be used for at least embedded computers, because the high software costs for these embedded computers were the reason why Ada activities were initiated about ten years ago.

Saab has since 1979 followed the Ada activities, and during the last two years we have studied Ada for usage in our products, which are embedded computers for on board use in space applications. The Ariane launcher, Hermes shuttle and Columbus, which will be the European part of the NASA Space Station, are examples of such applications.

On board computers, OBC:s , have been developed by Saab since 1973, and these OBC:s are used in a number of applications, for example the Ariane launcher, the EXOSAT, SPOT and HIPPARCOS satellites and in the EURECA platform. Up to now, assembler language has been the main language for these embedded computers even if there are high level language compilers as Pascal, Coral66 and Fortran available.

Our on board computers are designed for use in space applications, where maintenance by man is impossible. All manipulation of such computers has to be performed in an autonomous way or remote with commands from ground. In a manned Space Station some maintenance work can be performed by service people on board, but there are still a lot of applications, which require autonomous computers, for example vital Space Station functions and unmanned orbital transfer vehicles. In other words, the aspects in this paper are most valid also for embedded computers in the NASA Space Station.

The rest of this paper will deal with those aspects, which have come out of the analysis of Ada characteristics together with our experience of requirements for embedded on board computers in space applications.

* Ada is a registered trade mark of the U.S. Government Ada Joint Program Office.

MOTIVES FOR Ada USAGE

There are at least two large groups, which perhaps have not exactly the same requirements on the programming language, these are the computer manufacturers and the customers.

Saab is manufacturer of embedded computers for space applications on board. We wish to make some form of profit as a result of our computer production. If our computers are very attractive also without Ada support, then it is tempting to avoid Ada, because the costs to develop an Ada support to a computer are very high. On the other hand, if it is possible to increase the price of a computer because the Ada support makes it more attractive, then it is an interesting alternative.

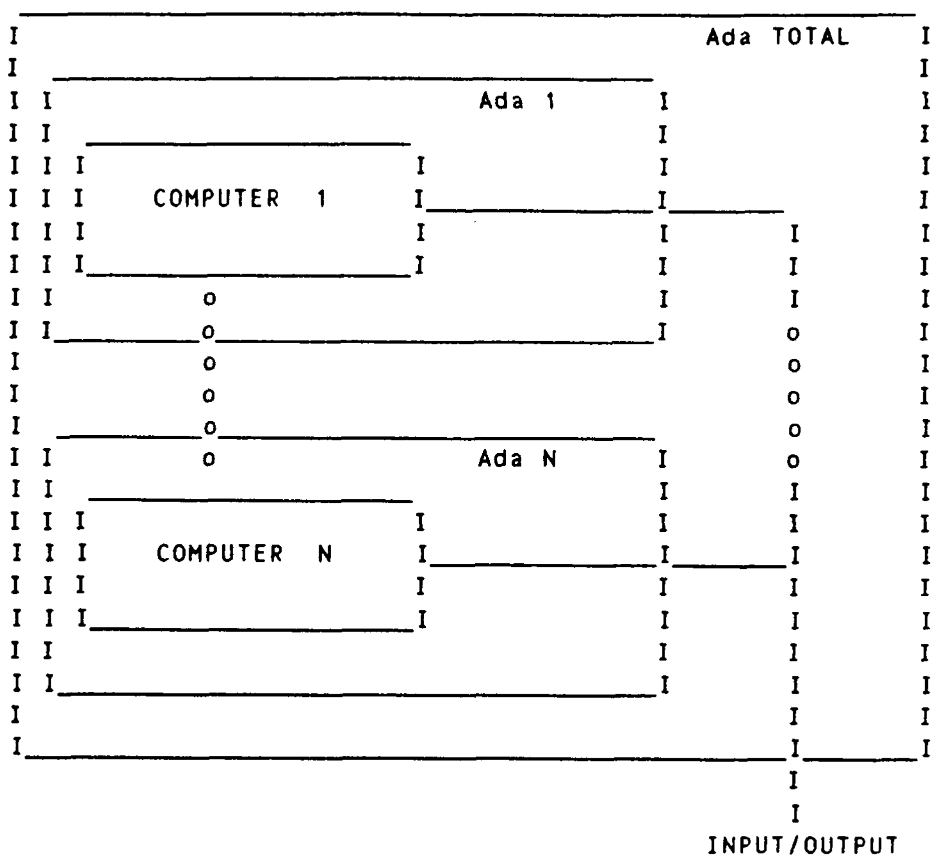
Ada is designed to give low maintenance costs and high quality of the program. Therefore, because the customer pays the life cycle cost, Ada will be attractive for customers with long lived projects, which contain vital functions. However, development of Ada programs requires at least the same or more resources compared with development in other languages, because Ada is designed to be "read" (maintenance) rather than be "written" (development). This means that for very short lived programs, another language can be a better choice. Most space projects have a very long life cycle, which means that Ada ought to be a good choice from space customer's point of view.

The most important impact on the Ada development, however, comes from large customers, who require Ada as programming language. Then the computer manufacturers have to give Ada support to their products if they want to participate in the project. This happens for example when Department of Defense requires Ada for their projects. The Swedish DoD also requires Ada after January 1986 for new military projects. The same thing happens in a number of countries. Because of these very strong "pro-Ada" forces, the usage of Ada will probably increase all over the world, and the computer manufacturers have to give Ada support to their computers if they want to have a chance in the competition on the market. The customer will choose a computer with Ada support, because it is important to minimize the software costs, which take such a great part of the costs of a computer system during the life cycle.

From programmers point of view, Ada is a nice language. It is a total language, which means that no dialects are required (or allowed). Ordinary operating system functions are, for example, included in the language. A normal reason why assembler languages have been used for embedded computers is the interface between the embedded computer and the external world. Very often this interface is a special non standard type, which is not supported by any high order language. Ada is designed especially for such embedded computers, which means Ada in fact is the first chance to get embedded computer software, which is possible to maintain in a pactical way. Because of the high degree of standardization in Ada, the risk for misunderstanding between programmers is minimized. This is important when a number of countries participate in a space project, and will contribute to an increased program quality and lower cost.

Ada PROGRAM LIMITS

View the following figure:



The Ada language tells nothing about the number of computers, which take part in execution of an Ada program. Suppose that computer modules 1 to N cooperate to execute one program, Ada TOTAL, then the programmer has to think only on the black box function and on the INPUT/OUTPUT signals. All communication and time synchronization between computer modules and task allocation to the modules 1 to N will be performed by a program, that can be generated by one Ada compiler. This compiler has to know a lot about each computer module and also about the communication lines between the modules. Another way to solve the problem is to write a number of programs, Ada 1 to Ada N , and use one or more compilers, which generate code for one computer module at the time. Here the programmer must write the program, that has to administrate the time synchronization and communication between the computer modules and also for the task allocation.

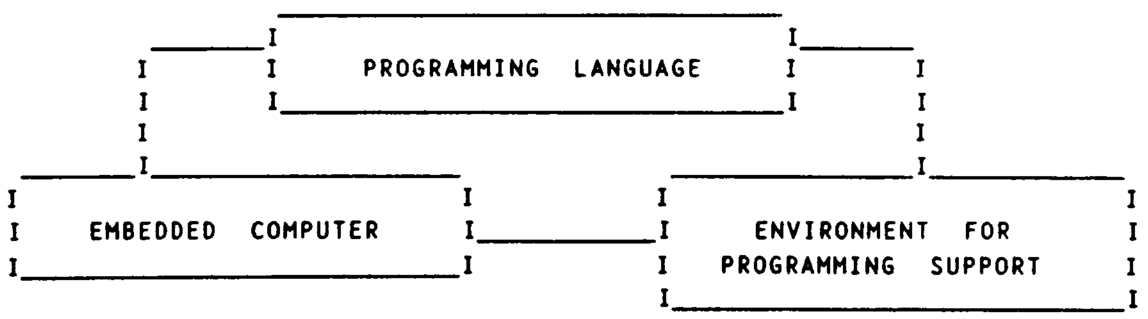
From the computer customer's point of view it seems attractive to choose the Ada TOTAL alternative and let the Ada compiler do a lot of the work. In the Space Station, for example, it would be nice if one "clever" Ada compiler handled the communication between computer modules. However, in practice there are a lot of problems with this Ada TOTAL alternative. Development costs for such a compiler will be very high and time very long; if the computer structure is very complex, then the risk is high that no compiler at all is available when needed. Even if such a compiler is possible to develop, the price will be so high, that it is difficult to sell the compiler. Its special architecture makes it impossible to use in other applications, and then the development costs can not be spread out on a number of products. In the Space Station project, the computer modules will most probably come from various computer manufacturers, and this complicates the Ada TOTAL compiler still more. Questions about responsibility, maintenance and modifications of the computers as well as the Ada compilers during the life cycle will be complicated when many companies are involved in and controlled by the same compiler. Also the support equipment for development, integration,verification and maintenance will be complicated and so will the administration of the support equipment.

The Ada TOTAL line seems to be not usable in practice because of the following reasons: too high complexity in technical functions and in administration between companies. This indicates that the other way with several programs, Ada 1 to Ada N , will be a better one. Then the interface between compiler and computer can be handled within one company. The interface between companies will consist of the communication lines. Experience shows that connection of computers from two or more manufacturers can take very long time and be extremely expensive if the communication interface is badly defined, because people think that they do understand each other, but they do not. Therefore it is very important to define all communication protocols in detail as soon as possible in a project.

In the Ada 1 to Ada N alternative, it is not possible, within the Ada concept, to distribute a data base to several computer modules. The solution will be to provide each computer module with a program, that will handle communication lines and distributed data between computer modules. By defining a suitable program interface to this communication interface, the application programmer will get a feeling of distributed data base.

Ada PROGRAMMING SUPPORT ENVIRONMENT , APSE

From software point of view,a computer system consists of: embedded target computer hardware and software, environment for development and maintenance of software, programming language for the embedded computer.



The programming language in fact belongs to the environment, but because of its great importance, it is often handled as a separate component.

It is important to choose and adapt these three computer system components to each other in a proper way to make it possible to reach specified requirements, for example quality and low life cycle cost.

Not only the embedded computer but also the programming support environment and language must work properly during the entire life cycle, which for space applications can be very long, perhaps 10 to 20 years. Consequently Ada, which is designed for long life cycle projects and easy to maintain, shall be used not only for the embedded computer but also for programming of the environment computers.

The Stoneman specification for Ada Programming Support Environment, APSE, says that the programming language for the APSE itself shall be Ada; only the most hardware near programs may be written in another language, normally on a lower level. Then, if the APSE hardware must be exchanged to another one, only these lower level programs have to be rewritten. However, these programs close to the hardware can be very hard to develop, which means that the software costs for exchange of APSE hardware can be very high, even if most programs are written in Ada. Rewriting of the APSE kernel programs will also affect the APSE quality in a negative way, because each time a new piece of program is included, the number of software design errors increases, and it takes time to reach the same quality level as before the rewriting. The following figure shows this.

```
PROGRAM QUALITY
     I
     I                                                   .
     I                                   .            .
     I                   .           .   .        .
     I               .   .       .       .    .
     I           .       .   .           .
     I       .           .
     I   .
     I___________________I_______________I___________________ TIME
                     REWRITING       REWRITING
```

It is possible to avoid these effects by using environment computers with either totally compatible or standardized instruction sets and I/O signals. Then no software at all need to be rewritten because of new computers in the environment, and the QUALITY/TIME diagram will have the following look:

```
PROGRAM QUALITY
     I                              .
     I                       .
     I                 .
     I            .
     I        .
     I    .
     I.
     I________________________________________________ TIME
```

It could be a risk that such standardization and compatible ideas destroy the possibilities for computer hardware evolution, but I do not think there is any risk in practice. An example of this is the MIL-STD-1750A. There are very powerful and usable such processors, even if their instruction architecture is old. Remember that the most expensive part of a computer system, the software, will be improved by this philosophy.

If the embedded computers could use the same instruction set as the computers in the environment, then we would get a number of advantages. The price of the Ada compiler for the embedded computer can be lower if the compilers to the environment and to the embedded computer can be derived from each other. This is important because Ada compiler development is very expensive. It will also be easier to find programmers for development and maintenace during life cycle if the instruction set is used also in many other computers. The same programs can be utilized in the environment as well as in the embedded computer, which is positive from quality point of view.

Software activities around on board space applications are different compared to ground applications. On ground it is possible, and also normal, to deliver software with guaranty and modify if it does not work properly during integration and validation. On board in space the software has to work the first time it is used in practice, especially vital programs for spacecraft control. Of course the flight software is carefully tested on ground before launch, but the real environment is met after launch, for the first time. The flight software has to pass several phases: first it will be developed on ground and this phase is perhaps the most "normal" one, but it is still different from program development for embedded systems on ground. For example, regard the debugging session, where an accepted method is to use in circuit emulation for embedded systems on ground. The principle is that control lines and busses are drawn out and the processor removed from the embedded computer to the programming support environment, from where it will be possible to control the embedded system in detail. When an on board computer for space use shall be validated or integrated in a subsystem on ground, it is very difficult to use the in circuit emulation method, because the control lines, busses and processor can not be drawn out because of quality, reliability and practical reasons. This means that the best software development tool existing today for embedded computer systems, the in circuit emulation, can perhaps not be utilized for embedded on board space computers when it is most needed. The on board computers, developed by Saab, are equipped with special hardware and microprogram software to allow powerful debugging also in these embedded situations. The next phase for the flight software is execution in space. For manned missions the maintenance can be performed on board, but the program debugging will be still more complicated than on ground. Connection of programming support environment for in circuit emulation is as difficult as on ground, but even if it would be possible, it can be difficult to bring the ground programming support environment into space. Probably specially designed programming support environment has to be developed for use on board in manned missions. For unmanned missions the maintenance has to be performed remote, which means that a number of other computers and communication lines constitute the interface between the embedded on board computer and the programming support environment. To make remote maintenance possible, the on board computer has to be equipped with extra programs and perhaps also extra hardware to perform debugging commands from ground. The programming support equipment as well as the embedded computer also must have the right interface to the intermediate computers and communication lines.

What does standardization mean for Ada Programming Support Environment, APSE? Regard as an example the debugging on ground and the remote maintenance on board as described above (also visualized in the following figure).

```
          GROUND                              ON BOARD
   _______________________          _______________________
  I                       I        I                       I
  I   EMBEDDED COMPUTER   I        I   EMBEDDED COMPUTER   I
  I_______________________I        I_______________________I
              I                                I
   ___________I___________          ___________I___________
  I                       I        I                  *    I
  I         APSE 1        I        I     INTERMEDIATE      I
  I_______________________I        I   COMPUTERS   AND     I
                                   I   COMMUNICATION       I
                                   I_______________________I
                                               I
                                    ___________I___________
                                   I                       I
                                   I         APSE  2       I
                                   I_______________________I
```

*
These computers may also contain a number of Ada programs, each supported by its own APSE.

Will it be possible to utilize APSE 1 from one manufacturer during the development phase and APSE 2 from another manufacturer for maintenance during mission? It ought to be possible if the flight software shall be supported during life cycle, which is the meaning when Ada is used for embedded systems. However, my experience of APSE is that as long as you are working inside the host computer, where the APSE software is executed, then APSE gives all necessary help. It administrates editing, compiling, printing and management of program versions for example. But when the target computer in the embedded system shall be reached for program loading, debugging and execution control, then APSE does not support that. Then you have to use another equipment, which is not involved in APSE. This is unsatisfactory, especially because the program debugging perhaps is the most difficult phase in the software life cycle. Also the philosophy of program loading into embedded systems is interesting. Suppose that the embedded computer has no natural way for program loading, then will this problem be solved as an application function, or is it such an ordinary problem for embedded computers, that it shall be specified for APSE?

Briefly the requirements on APSE for embedded computers on board in space applications are:

- -use Ada also for the APSE computers
- -use an instruction set in APSE computers, which will not be changed during life cycle
- -use the same instruction set in APSE computers as in the embedded computer to give maximum support to the embedded on board computer system
- -APSE shall allow powerful debugging of the embedded computer on ground specially during integration and validation phases, when internal signals are difficult to reach
- -APSE shall be possible to bring into space for manned missions, e.g. Space Station
- -APSE shall allow powerful debugging of the embedded computer on board in space to be performed in a remote way from ground
- -APSE shall be standardized in that way that it is possible to maintain the same Ada program on board from various APSE:s
- -APSE shall comprise all tools, which are necessary for programming support of the embedded on board computer. At remote debugging, however, the intermediate computers will not be seen as part of APSE, but rather as part of the communication line
- -An APSE contains a lot of functions, which are intended to give, for example, high quality programs and low life cycle cost. Some of these functions require that the programmer follows given rules in order to reach the goal. Here is a risk for conflicts between these rules and the working routines of a company. No big company will modify the organisation and working routines to fit an APSE. Therefore a very important requirement for an APSE is flexibility to make adaption to various company organisations and working routines possible.

The APSE also ought to contain a Program Design Language, PDL, to support the early software phases before coding as well as the maintenance phase, when it can be a good help for understanding of program function. Ada has been tested as PDL at Saab, and been found to be a good candidate for that job. It is practical to use the specification part of an Ada program during program design, because in that way a part of the program code exists when the coding phase begins.

Ada COMPILER

At Saab we are looking for Ada compilers, which generates usable code to processors, which we can use in our embedded computers on board in space applications. Until now, many of our on board computers have been equipped with special instruction sets, which have been adapted to each specific application in order to make it as powerful as possible. This has been possible because the requirement of programming support has been mainly an assembler and a debugger, which are possible to redesign for each project. Now, when Ada will be required, we have two ways to go. We can develop Ada compilers to our Saab computers with special instruction sets. We can also use microprocessors, which already are equipped with Ada compilers, in our on board computers. Because of the high costs, which are required for own development of Ada compilers to a relatively small number of special purpose computers, it is tempting to try to use more standardized microprocessors with already existing Ada compilers. However, there are a number of important aspects to think about when buying an Ada compiler, which will be used for on board computers in space applications.

Often you read in reports and other papers that it is important to use "commercial microprocessors" and "commercial software" in order to keep costs on a low level. Our experience says that this is not always true for long life cycle projects, for example in space projects. The manufacturers of commercial microprocessors and software have to compete with each other about the commercial market, where the big money are. They offer new products within short time intervals in order to try to be one step before the other. The manufacturer's interest of maintenance for the old product will decrease, of course, because he has to spend all resources on the latest product. If you use such a product in a project with long life cycle, you will have to choose between: A) keep the old product, which probably is space qualified, and hope that it will be supported during the life cycle of your project; or B) take the new product and hope that it becomes space qualified. No one of these alternatives is good. Each of them can end up in very high costs to find a usable alternative. A better alternative would be to define and standardize a processor for space applications on board. It could be done analogous with the MIL-STD-1750A, defined by DoD. If such a standard is used in space projects with long life cycle, then you have always a good chance to find a new qualified component if you lose one. You can also replace such a processor with a new one with no software modifications at all. The old Ada compiler can for example be used. It is a hard job and it takes long time to define and state such a standard,and until that is done I think a good idea would be to use the MIL-STD-1750A for embedded computers in space instead of using commercial processors and software.

When Ada is used as programming language, it would be an advantage if a future processor standard for space were adapted for Ada. The MIL-STD-1750A specifies a register machine, while a stack machine architecture would be a more suitable processor standard in space applications where Ada is used. The reason is that the instructions of a stack machine are on the same level as Ada statements, while the instructions in a register machine are on a lower level. This gives in turn as result that the stack machine requires smaller program memory compared with the register machine. Comparisons in practice show that a stack machine needs only about half of the memory required by a register machine. The stack machine probably has higher performance too, for example because of fewer accesses to memory and possibility to parallel processing of the high level instructions inside the processor. Performance tests often consist of execution of an instruction mix. The instructions are normally taken from the instruction set of a register machine, which is a drawback for the stack machine. In this way the register machine can give best test result, while the application function is best performed by the stack machine. It is therefore of great importance that the performance tests are specified in functional terms in order to find the most suitable computer.

Small memory and high processing power are perhaps not important requirements in embedded systems on ground, but it is in space applications on board. The reasons are: very high costs to put power, mass and volyme from ground into space, and computer reliabilty decreases with memory size, which means that it is very valuable if you can do the job with one processor instead of two and also if the memory is small. The ground expression "waste with memory, it is cheap!" is not true for on board computers in space; normally a qualified memory for space costs one hundred times more than a ground memory.

Also the Ada compiler has great influence on the memory size and processing power. However, most of the Ada compilers available today are not well adapted for embedded computers in space applications. In fact most Ada compilers are not developed for embedded computers at all, but for large administrative computers. Then some of them have been adapted to various embedded computers. It is a risk that small memory and high performance have got low priority during development of such a compiler. Probably few users of ground systems are interested in compact code and high performance of the generated code. Execution speed of the Ada compiler itself is often more interesting. Because of this I think that development of special Ada compilers, in order to meet the requirements for embedded computers in space, is motivated. The development costs would be payed back very soon because of lower costs for embedded computers on board. It seems natural for me to generate effective code for the embedded system on board with an Ada compiler on ground instead of increasing embedded computer resources on board because of ineffective code produced by an Ada compiler on ground. A standardized instruction set for embedded computers on board in space should also contribute to the generation of more powerful code in that way that the Ada compiler manufacturers could concentrate their efforts on optimization of code to only one instruction set; today, when they have one Ada compiler working, it is time to start development of an Ada compiler for the next instruction set. There is no time for improvement work. Most Ada compilers are developed by software companies, while the processor manufacturer, who has the best knowledge about the processor and instruction set, would be most suitable to develop an Ada compiler, which generates optimal code. A standard instruction set would give also the software companies this possibility.

Normally the Ada compiler does not generate all the code to an embedded computer program at each compilation. A large part of the program is involved in the Runtime Support Library, RSL, which has been created earlier. The RSL contains computer specific programs and perhaps also real time programs, e.g. ordinary operating system functions, which will be called by the code generated by the Ada compiler. RSL can be a relatively large program package, perhaps 100 - 200 kbytes. It is therefore important that RSL is developed in a modular way, which allows generation of small Ada programs. If Ada shall be used also for small embedded computers, which was the intention when the Ada work started more than ten years ago, then it must be possible to sort out RSL pieces of about 10 kbytes, or perhaps smaller. That selection job shall be performed by the Ada compiler automaticly. The RSL is highly dependent on the hardware architecture of the embedded computer, which means that you normally have to modify the RSL as soon as signal lines or components are changed in the hardware. This is a great problem from validation point of view.

Validation of an Ada compiler is a costly process, therefore it is an advantage for a compiler manufacturer if his Ada compiler crn be used in as many applications as possible. For administrative computer system, e.g. a VAX with line printers, disks and terminals, a validated Ada compiler can be used in many installations without to be modified. This is possible because of the standardized input/output units. For embedded computers it can be very difficult to find even two computers with equal i/o interfaces. This gives different RSL:s or different Ada compilers, which in turn means separate validations for each system. Ada Joint Program Office, AJPO, proposes easier rules, they will accept modification of validated Ada compilers and call such a compiler for a derived Ada compiler. I think this gives a not desired result.

One big reason to start up Ada development was wishes to decrease the high software costs for embedded computers, and these high software costs depends to a very high degree on all these complex i/o interfaces. Therefore, usage of a great number of modified Ada compilers in order to fit all these i/o interfaces, will not decrease the software costs for embedded computers. A better idea ought to be usage of a smaller number various i/o interfaces, which have been standardized, in order to make it possible to use one validated Ada compiler to many embedded computer installations. Improvements of a product can be done inside these i/o interfaces. It is not always necessary to modify also the interface, even if it seems as a good idea from technical point of view.

During the validation of an Ada compiler no measurements are made about size of generated code or performance of that code. This is a lack if the code shall be used in an embedded computer, because applications in such computers are often time critical, for example to take a sample of an analogous signal each 10 millisecond with an accuracy of 1 millisecond. The situation can be that the required function can be performed by the code from one validated Ada compiler, but when you have to exchange your Ada compiler, for some reason, to another validated Ada compiler, then the function can perhaps not be performed by the new code. Analogously the memory can hold the code from one validated Ada compiler but perhaps not from another. I think that the validation tests should be supplemented with code size and performance tests, at least for Ada compilers to embedded computers on board in space applications.

Generally, time is an important parameter for embedded computers on board in space. A normal requirement is that a number of computers on board have to be synchronized to an absolute time. If the code to these computers are generated by different Ada compilers, then time synchronization has to be performed via the communication lines, if it shall be performed by software. The requirements on the time accuracy are often so hard that it is impossible to perform synchronization by using the long way via the application communication protocol. Instead lower level protocols have to be used. If this protocol software is included in the RSL or generated by the Ada compiler, then the Ada compiler must be seen as affected by the time synchronization requirements, but hopefully the different Ada compilers do not have to exchange information to each other.

A question, which arises very often is whether it is possible or not to combine Ada programs to programs written in other languages. The link process can be organized to handle that, but I think it is better to translate source code from other languages into Ada source code in order to get all advantages from the Ada compiler.

Ada IN MULTIPROCESSOR COMPUTERS

Writing an Ada program to a multiprocessor computer shall from the programmers point of view be equal to writing the program to a single processor computer. Static allocation of tasks between various processors is a job for the Ada compiler and is transparent for the programmer. If dynamic allocation shall be possible, the necessary programs to do this must be generated by the Ada compiler or be included in the RSL. During program loading and debugging also the APSE has to handle all processors in such a way, that they are transparent for the programmer. The RSL, Ada compiler and the rest of APSE will all become more complex compared with a one processor computer and are because of that more expensive to develop.

Fault tolerant computers can be seen as a kind of multiprocessor computers, which have possibility to detect faults and move tasks from a faulty unit to a fresh one. As for other multiprocessor computers, it shall be possible for a programmer to write an Ada program without thinking on the fault tolerant computer architecture. It is a job for the Ada compiler to generate necessary programs, or they may be included in RSL. For example, if the requirements are Fail Operational/Fail Operational for a computer, then the RSL or the code generated by the Ada compiler must be able to take care of two faults and still keep the Ada application program executing. As mentioned above, it is very costly to develop support software, e.g. Ada compiler, RSL, loader and debugger, to this type of computers, and therefore it desirable to use as many as possible of equal computers in order to utilize the costly support software. The work to detect faults and switch to fresh units can be performed either by hardware or by software. The reliability will often be better, if software is used, because of less hardware, and because of that the software method is attractive for space applications.

An interesting question arises when you are talking about n-version programming in order to be tolerant against design failures. If all n versions of a program are written in Ada, then n different APSE:s ought to be used for code generation and debugging, because such a complicated software package as APSE probably contains design failures too. Then the question will be: is it realistic to work with n different APSE:s from economical point of view?

N89-16347

COMPARING HOST AND TARGET ENVIRONMENTS FOR DISTRIBUTED ADA PROGRAMS

MARK C. PAULK
SYSTEM DEVELOPMENT CORPORATION
4810 BRADFORD BLVD NW
HUNTSVILLE, AL 35805

Abstract

The Ada* programming language provides a means of specifying logical concurrency by using multitasking. Extending the Ada multitasking concurrency mechanism into a physically concurrent distributed environment which imposes its own requirements can lead to incompatibilities. These problems are discussed. Using distributed Ada for a target system may be appropriate, but when using the Ada language in a host environment, a multiprocessing model may be more suitable than retargeting an Ada compiler for the distributed environment. The tradeoffs between multitasking on distributed targets and multiprocessing on distributed hosts are discussed. Comparisons of the multitasking and multiprocessing models indicate different areas of application.

Keywords: Ada, distributed processing, multitasking, multiprocessing, Ada Programming Support Environment (APSE), software engineering, computer networks, interprocess communication.

1. INTRODUCTION

In designing a solution to a real-world problem, the systems analyst is frequently faced with the fact that the real world functions in terms of concurrent activities. Many applications are modelled most naturally by logically concurrent tasks, but most computer languages do not support concurrency. Even when concurrent activities can be distributed on a computer network to achieve physical as well as logical concurrency, the designer must build the interfaces between the physically distributed components of the system as well as partition on its logically concurrent boundaries.

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

Recognizing that concurrency is the natural approach to solving many problems, the Department of Defense (DOD) developed multitasking as an integral part of the Ada programming language specification. Concurrent tasks may communicate through task activation and termination; they may share global variables; or the communicating tasks may rendezvous using entry calls and accept statements. Synchronization between communicating tasks may use selective waits, conditional entry calls, or timed entry calls [1].

The host/target model was used in designing the framework for Ada environments [2,3]. Ada programs are developed on a host computer system. The program is compiled by a cross-compiler, and the executable module downloaded to the target system on which it is to execute. This model describes the typical softwared development environment for embedded systems.

A range of options can be considered for the distributed target. Alternatives to multitasking may be chosen, such as a multiprocessing approach relying on an I/O-oriented interface for interprocess communication. Fully transparent distribution of the program can be implemented, or, as is more common for most efforts, only tasks can be distributable. Although the Ada multitasking model intuitively seems to be the natural model for concurrency in the distributed environment, Cornhill has suggested that the Ada programs could and should be arbitrarily distributable [4,5]. Packages and individual blocks of code as well as tasks should be distributable. Ada programs should be developed using the Ada multitasking model for logical concurrency regardless of the underlying physical concurrency. The physical distribution of the Ada program can be specified using a distribution language which is input to the compiler with the Ada source code. The tradeoffs between the various alternatives must be carefully considered before an approach to implementing distributed Ada programs is selected.

The terms "task" and "process" are frequently used interchangeably. In this paper tasks are independent but interacting program components which execute in parallel. A process is an independent program execution and its context. It is the basic unit scheduled for execution by the operating system and represents the execution of a program [6]. A single Ada program may contain many tasks, yet execute as a single process under an operating system which runs many concurrent independent processes.

2. DISTRIBUTED PROCESSING REQUIREMENTS

Distributed processing may be implemented on radically different types of architecture. Shared memory architectures have multiple processors sharing one or more global memories, or processors with local memory may be interconnected by message-oriented communications links. These message-oriented links may be strictly point-to-point, or they may have a broadcast or multi-drop capability.

Distributed systems may interface by messages, remote procedure calls, rendezvous, monitors, or shared variables to name a few of the approaches. At the most fundamental level there are only two classes of communication technology: those which copy data, e.g., an I/O-oriented approach, and those which reference shared data, e.g., using global (shared) memory. Interrupts may provide an asynchronous change of control flow to signal an event or message exchange, or the message exchange may be referenced synchronously within the process. Various communications methods may be layered on these basic technologies to provide different access techniques and control flow structures.

There are a number of desirable capabilities for a distributed processing system. These include:

- support for multiple readers;
- support for multiple writers;
- support for multiple independent message streams;
- asynchronous input, i.e., a non-blocking receive;
- asynchronous output, i.e., a non-blocking send;
- support for locking shared memory data structures for mutual exclusion;
- control over the scheduling discipline;
- access to a system clock;
- an interval timer which can asynchronously signal events;
- control over the distribution of processes on the network;

- fault detection and damage assessment.
- transparent fault tolerance;
- support for multicast;
- support for broadcast;
- security features such as encryption.

Although a feature may be desirable, it may be impractical to implement for performance reasons. There is a trade-off between performance and desirable features such as fault tolerance that is application dependent.

Considerations in the area of distributed processor management [7] include

- the allocation of processors: static, dynamic, user-defined, or automatic
- the atomicity of distribution: packages, tasks, or procedures
- possible remote operations: rendezvous, activation/termination, remote procedure calls, and global variables
- remote dependencies and exception handling
- general network topics such as encryption, protocols, and fault handling.

There are two extremes to using Ada in the distributed environment. One extreme transparently distributes Ada programs across the distributed environment. There are, however, inherent problems in the Ada model of concurrency when applied to the distributed environment. Although solutions may exist to many, if not all, of these problem, the performance penalties extracted may render the multitasking model impractical.

The other extreme follows the multiprocessing model in which separate sequential programs are developed which can be concurrently executed. Ada programs can use the techniques developed during years of research into distributed processing issues. The drawback is the loss of the advanced software engineering concepts intrinsic to the Ada concurrency model. The advantage is that the system designer is explicitly aware of the underlying distributed architecture.

3. DISTRIBUTED ADA PROGRAMS

Implementing the Ada concurrency mechanisms on a distributed system is not a straightforward matter. A number of issues which are of concern in distributed processing are not adequately addressed by the Ada multitasking capabilities, and a number of assumptions implicit in the definition of Ada tasks do not necessarily hold true in the distributed environment [8]. The implementation of physical concurrency may place restraints on the design of logical concurrency, for example, the use of global variables in the absence of shared memory. These constraints may be driven by both performance and feasibility restrictions.

The Ada Language Reference Manual indicates that multitasking can be transparently implemented on a distributed system [9]. Several features of the language, however, imply a single-memory system [10]. Although entry calls and accept statements are the primary means of synchronization of tasks, and of communicating values between tasks, the use of shared variables is also described in the language specification. Global variables imply a common memory. Access objects as rendezvous parameters imply a common memory. Many distributed systems, however, do not support shared memory.

Connection management is not supported. There is no suitable language construct to represent a node in the network; therefore distribution of the program cannot be handled from within the language.

All possible constraints on synchronization cannot be expressed using the rendezvous primitives. The rendezvous provides synchronization points for communicating tasks. Ada provides only synchronous communication (other than through shared variables). Asynchronous communication implies nonblocking sends and receives. This problem can be addressed by inserting a buffering task (also called agent tasks [11]) between the sender and receiver, but this may impose a significant degree of overhead.

Conditional entry calls imply that it can be quickly established whether the called task has executed the accept and that the queue is empty. Since the delay statement would be used if a "timed" response was adequate, conditional entry calls will be used by tasks that cannot tolerate excessive delay. When the called task is on a remote node, timely response becomes a critical - and unquantified - issue.

Timed entry calls may imply a potential race condition between the rendezvous and the timeout. Should timeout be measured from the calling or accepting tasks involved in the rendezvous? If from the calling task, as seems logical, race conditions may occur where the calling task has aborted a rendezvous that the accepting task has initiated. If from the accepting task, are the semantics of the language preserved?

An interval timer capability is not supported. The Ada delay statement guarantees a minimum delay; the actual time interval can be arbitrarily longer than that specified by the delay statement and still satisfy the semantics of the delay.

Packages STANDARD and SYSTEM need multiple definitions in a heterogeneous distributed environment. This implies an interface to the network presentation layer and possibly a canonical representation of entities. Assumptions in target-dependent representation clauses may imply a specific system in a heterogeneous environment.

Fault tolerance is not addressed [12,13]. What happens when a distributed system has a processor crash? Can a "shadowing" task take over the functionality of a "dead" task? Can the system degrade gracefully? Ada makes no explicit provision for continuation. When a processor failure occurs, services and data may be lost; tasks may be permanently suspended on the surviving processors; and the context of some tasks may be lost. A replacement task cannot assume the name of the task it is intended to replace, and there is no provision for redirecting the communication path used before the failure.

Using Ada in the distributed environment may require extensions to the language [12], which, by definition, means the language is no longer Ada. If there are restrictions on what Ada constructs are distributable, i.e., shared variables are not permitted, can the compiler be validated? If the compiler generates full Ada for a uniprocessor and a subset for a distributed target, can it pass validation as a derived compiler based on its uniprocessor mode? The issue of validating Ada compilers for distributed environments is not resolved at this time. By one philosophy each host/target pair must be validated. Although validation policy has evolved beyond that point, the question of a distributed architecture on validation is debatable.

One way of avoiding the entire validation issue and the problems of distribution is to not support physical concurrency in the compiler. Traditionally, distributed computer systems have applied some variation of multiprocessing.

4. ADA AND MULTIPROCESSING

Multitasking enters an area traditionally considered the province of the operating system. In attempting to define the Ada host/target environment, the Stoneman document specifies an Ada Programming Support Environment (APSE) to provide a framework for writing Ada programs [2,3]. Examining the boundaries between an APSE and the target system reveals several related areas: the Ada language, the run-time system, the operating system, and the programming support environment. The Kernel Ada Programming Support Environment (KAPSE) provides access to the operating system routines. An APSE provides a multiprocessing host environment for software development. The target's run-time system provides the virtual machine on which an Ada program runs. Issues which are not specified in the Ada language definition and must be addressed by the run-time system include the broad categories of job scheduling, memory management, security, fault tolerance, and distributed systems.

In an APSE tool composition implies a need for one Ada program to invoke another completely separate Ada program [14]. Since the Ada language has no such facility, support for tool composition must be supported by the KAPSE. An INVOKE_PROGRAM primitive can suspend the calling program, execute the called program to completion, and then resume the calling program. The primitive can also be non-blocking.

The Common APSE Interface Set (CAIS) attempts to provide a standard host environment for developing host tools [6]. The CAIS includes both process initiation and interprocess communication mechanisms. The distributed environment, however, is a deferred topic under the proposed MIL-STD-CAIS. If the CAIS is extended to address the distributed host environment, applying the same mechanisms to the distributed target is straightforward. The distinction between host and target systems is largely artificial for this instance.

Research in distributed systems has explored many avenues for implementing concurrency including multiprocessing and integrated approaches similar to multitasking. The most significant problem with the Ada and multiprocessing approach is that it discards the software engineering concepts central to the language. The strong type checking and information hiding capabilities integral to Ada are seriously compromised by using message-oriented mechanisms.

Part of the Ada design philosophy is that modularity and abstraction are well-proven means to overcome natural human limitations in dealing with complexity. Should a system designer be aware of an underlying distributed system? To provide the time-critical performance required by the application it may be essential that the designer have explicit understanding and control of the distributed system. In other systems which do not have real-time requirements it may be irrelevant to the system designer how the underlying hardware implements the design.

A compromise between these approaches is to develop a pre-processor which takes as input a single multitasking Ada program and outputs multiple Ada programs (one per node) that use site-specific mechanisms for interprocessor communication [15,5]. Such a hybrid approach would provide a portable tool for building distributed Ada programs. The pre-processor could be written in Ada, accept an Ada program as its input, and output a set of Ada programs which could then be compiled for the appropriate target. The pre-processor could use a standard software communications package which provides a basic message-oriented networking capability. This package could be reimplemented for a given distributed architecture without changing the pre-processor. Proxy tasks could then be used to handle rendezvous between nodes.

5. CONCLUSIONS

In a real-time embedded target environment the expense and complexity of implementing an efficient Ada compiler for a given distributed architecture may be a comparatively minor issue. A distributed system could be built incorporating solutions to the problems with distributed multitasking which have been discussed. Whether such a system could provide adequate response in a hard real-time environment is questionable unless the compiler is customized for a specific distributed target.

Using the multiprocessing approach requires knowledge of the distributed architecture at system design. This is not necessarily bad, but current work in designing distributed computing systems emphasizes deferring a binding of the system to the architecture. The host environment, as opposed to the target environment, requires an interprogram communications mechanism to aid in tool composition. The extension of such a mechanism for the distributed environment can provide a portable distributed processing capability.

Combining multitasking and multiprocessing may be the most promising approach, but the basic problems in distributing Ada programs must still be addressed. For real-time environments the designer must remain aware of the performance implications of design decisions.

6. REFERENCES

1. Paulk, M.C., "Interprocess Communication in Ada," Proceedings of IEEE Southeastcon '84, April, 1984, pp. 33-35.

2. "Requirements for Ada Programming Support Environments: Stoneman," Department of Defense, February 1980.

3. Buxton, J.N., and Druffel, L.E., "Requirements for an Ada Programming Support Environment: Rationale for Stoneman," COMPSAC 80, October, 1980, pp. 66-72, reprinted in The Ada Programming Language: A Tutorial, ed. S.H. Saib and R.E. Fritz, IEEE Computer Society Press, 1982, IEEE Catalog No. EHO 202-2.

4. Cornhill, D., "Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets," Proceedings of the 1984 IEEE Conference on Ada Applications and Environments, pp. 153-162.

5. Cornhill, D., "A Survivable Distributed Computing System for Embedded Application Programs Written in Ada," ACM Ada Letters, Vol. 3, No. 3, Nov/Dec 1983, pp. 79-86.

6. Proposed Military Standard Common APSE Interface Set (CAIS), 31 January 1985.

7. Lomuto, N., Rajeev, S., and Grover, V. "The Ada Runtime Kit (ARK)," IEEE Real-time Systems Newsletter, Vol. 2, No. 2, Summer 1984, pp. 27-33.

8. Paulk, M.C., "Problems with Distributed Ada Programs," Proceedings of the 5th Phoenix Conference on Computer and Communications, 1986, pp. 396-400.

9. ANSI/MIL-STD-1815A, The Ada Programming Language Reference Manual, American National Standards Institute, 1983.

10. Dapra, A., et al, "Using Ada and APSE to Support Distributed Multimicroprocessor Targets," ACM Ada Letters, Vol. 3, No. 6, May/June 1984, pp 57-65.

11. Hilfinger, P.N., "Implementation Strategies for Ada Tasking Idioms," Proceedings of the AdaTEC Conference on Ada, October, 1982, pp. 26-30.

12. Knight, J.C., and Urquhart, J.I.A., "On the Implementation and Use of Ada on Fault-tolerant Distributed Systems," ACM Ada Letters, Vol. 4, No. 3, Nov/Dec 1984, pp. 53-64.

13. Knight, J.C., and Gregory, S.T., "A Testbed for Evaluating Fault-Tolerant Distributed Systems," submitted to Proceedings of the 14th Conference on Fault-Tolerant Computing Systems, June, 1984.

14. Stenning, V., Froggatt, R.G., et. al., "The Ada Environment: A Perspective," IEEE Computer, Vol. 14, No. 6, June, 1981, pp. 26-36.

15. R.A. Volz, A.W. Naylor, et al., "Some Problems in Distributing Real-time Ada Programs Across Machines," Ada in Use, Proceedings of the Ada International Conference, May 1985, issued as ACM Ada Letters, Vol. 5, No. 2, Sept/Oct 1985, pp. 72-84.

N89-16348

An Evaluation of Ada* for AI Applications

David R. Wallace, Intermetrics, Inc.

1. Abstract

Expert system technology seems to be the most promising type of AI application for Ada. An expert system implemented with an expert system shell provides a highly structured approach that fits well with the structured approach found in Ada systems. The current commercial expert system shells use Lisp. In this highly structured situation a shell could be built that used Ada just as well.

On the other hand, if it is necessary to deal with some AI problems that are not suited to expert systems, the use of Ada becomes more problematical. Ada was not designed as an AI development language, and it is not suited to that. It is possible that an application developed in, say, Common Lisp could be translated to Ada for actual use in a particular application, but this could be difficult. Some standard Ada packages could be developed to make such a translation easier.

If the most general AI programs need to be dealt with, a Common Lisp system integrated with the Ada environment is probably necessary. Aside from problems with language features, Ada, by itself, is not well suited to the prototyping and incremental development that is well supported by Lisp.

2. Is Ada Suitable for AI Development?

In order to answer this question we must look at what is required for developing AI applications.

2.1 AI Development Requirements

Two key phrases that describe AI development are:

- rapid prototyping
- iterative feedback development

AI systems are generally developed incrementally, where at each stage the current behavior is observed in order to determine exactly what the next stage should be. This requires great flexibility and is best supported by a language that allows either incremental compilation or interpretation. More specifically, AI development often requires heuristic search techniques that must be developed

*Ada is a registered trademark of the U.S. Department of Defense (AJPO) on-the-fly to match the particular problem at hand.

This type of development further requires flexible dynamically changing data structures without strong typing. Any use of data declarations must be either very limited or automated in some way. Otherwise, the overhead of constantly modifying these declarations becomes unacceptable and what is worse, error prone. This data problem has been dealt with very successfully in Prolog and Sail by the use of an associative data base. This allows data access without explicit knowledge of the surrounding structure.

In larger AI systems the concepts of data abstraction or object oriented programming are used extensively. See [7] for some background on object oriented programming in AI. The motivations for their use in AI is the same as that elsewhere: use higher level concepts and hide implementation details in order to make development, modification, and maintenance easier. Object oriented programs have algorithms and data that are very closely coupled. In AI applications this coupling can be very dynamic, having procedure values mixed in with the data the procedure is going to use. This makes it very easy to create very powerful parametrized transformations. How a large data structure is transformed can often be determined by values within itself. Much of the current uses of data abstractions in AI code written in Lisp is somewhat unstructured. This is partly since Lisp does not support data abstraction as a language feature. However, data abstraction use is becoming more formalized via the increased use of expert system shells (see below).

Most AI applications require some type of general value or attribute evaluation and propagation mechanism. A simple example of this is the parameters and variables in a Prolog program. The order in which these attributes are evaluated and propagated is dynamically determined; thus it is impossible to predict their storage requirements or lifetimes. This requires a very general storage management system with garbage collection. Data on a stack will in general have the wrong lifetime and data on a heap without garbage collection will overflow during most AI applications.

2.2 Ada Features Favorable to AI Development

Ada is a modern programming language providing clear and up-to-date control and data structuring facilities. Thus it should be very good at providing programming support for well understood and highly structured programming tasks.

Compared with other languages of its type Ada also provides a great deal of leverage in dealing with data abstraction and certain types of variability. The key features that support this are overloading and generics. Packages with overloading and generics provide a very powerful data abstraction mechanism. Such features allow what appears to be one procedure to deal with a number of different data types.

Another Ada strong point is its comprehensive support for modularity. The package concept is a very useful way of organizing a data abstraction. With the cross checking provided by the compiler it is very easy to divide a large task into modular pieces that can be developed independently and reliably.

Ada is highly suited to any task that is highly structured, has a relatively static behavior, and has a close correlation between control structure and storage lifetimes. There are probably some AI applications that fit these requirements.

The package concept allows the construction of what are the equivalent of Ada language extensions – in terms of data abstractions. This means that predefined library packages could be constructed to model the following:

- Lisp list-processing language features, see e.g. [3]
- associative database language features, see e.g. [4]

Such features would go a long way in allowing reasonable AI programming in Ada. However, there are potentially serious problems in implementing these packages appropriately (see below).

2.3 AI Problem Areas for Ada

Ada is unsuitable for dealing with the variety of problems and approaches arising in AI research applications.

2.3.1 Compilation

For the most part Ada requires compilation. For the purposes of AI development the lack of a reasonably fast interpreter or incremental compilation system is a very serious problem. Dynamic debugging in this environment is often used to determine the next stage of development. Without a fast interpreter it is very difficult to get an appropriately dynamic debugging system. In AI development, incomplete programs are often run with values supplied through the debugger when missing sections are reached.

The strong typing and the large declaration overhead add a very high cost to the iterative feedback loop used for AI development.

There are further problems caused by the use of a language that requires compilation when a large system is under development; that is ***recompilation.*** A small change in one part of a large system may (and often does) force recompilation and modification in most modules of the system. In a Lisp development environment the use of an interpreter eliminates the need for recompilation and the flexible and general data structures eliminate the need for rewriting data declarations.

To be fair it should be noted that Ada is much better than most other languages (like Pascal or C) in this area. Ada provides for modular consistency in a large system with both recompilation analysis and intermodule type checking. And further, Ada's support for data abstraction, even though somewhat static, allows for limiting the global effect of local changes.

It should be noted here that Ada systems that support incremental compilation are just starting to become available; see e.g. [2]. Such a system could go a long way toward alleviating these development problems.

2.3.2 Storage Management

As mentioned above under AI requirements, AI applications evaluate and propagate values or attributes in a very complex and often unpredictable manner. In any case, it is rare that the lifetime of these attributes follows the control structure of the program. This requires a system of managing memory independent of the stack mechanism. Direct user control of such a system (e.g. explicit FREE) is out of the question because of the certainty of error. In any real AI application it is also not practical to simply avoid deallocation; no matter how much memory is available it will be used up. This means there must be a sophisticated memory management system with garbage collection. This provides correct reclamation of storage when data lifetime is over. It is unlikely that Ada systems will provide such a feature because its high overhead conflicts with real time requirements. However, it should be pointed out that the Ada definition does not preclude garbage collection, see section 4.8 of the Ada reference Manual. This is a feature that could be associated with a pragma.

User defined garbage collection would require the creation of a storage exception that, when raised, would call a user subprogram to deal with it. This subprogram would need to use unsafe practices to do low-level heap manipulation and bookkeeping. Ada does not have the language features to allow higher-level control of storage for garbage collection. This is due to problems with its data abstraction capability which is discussed in the next section.

2.3.3 Existing AI Packages

One further problem with Ada, especially for near term use, is the lack of existing AI packages. There are, of course, many existing AI packages written in LISP.

2.4 Is Ada Suitable for AI Re-implementation?

If we assume some AI system has already been developed in an existing AI language, then we could consider translating it to Ada. This would avoid the problems mentioned above with the AI development cycle. Further, this approach has been used in a number of AI applications. There is a hazard here, however, since it may not be possible or practical to translate all AI systems to Ada. Translation problems can be mitigated by using AI-language coding standards to limit hard-to-translate features and usages. However, hard to translate features and usages are legitimate and necessary for some applications. Translation problems are likely to arise in two areas:

- data abstraction usage
- garbage collection

Garbage collection was discussed in the previous section. In general, a Lisp program using the full data lifetime capability will not be translatable to Ada.

Ada does not have a true data abstraction facility. Even though Lisp does not support data abstraction as a language feature per se, its flexibility allows the user to define and use powerful data abstractions. Ada supports encapsulated data types via the PACKAGE feature, but does not provide explicit abstract type construction features. This will create translation problems. Missing functions or features include:

- *updating structures within the package to reflect the instantiation of an object:*
  Ada does allow auxiliary structures within a package but there is no automatic way to coordinate it with object creation. Such use is necessary, for instance, to do storage allocation with garbage collection.

- *type instantiation parameters or run-time type attributes:*
  For instance, a user cannot create a string type with string-length as a type-attribute.

- *initialization and finalization of an object:*
  These are necessary when data types interact with their type context. For example, in the case of garbage collection, it is necessary to record information both when an object is allocated and when it is de-allocated. Ada only allows a limited form of initialization; i.e. when the data representation

is a record structure. However, there is no way to do finalization.

For more details on the abstract type problems of Ada see the SRI analysis of Ada for AI uses, [5].

2.5 Expert System Shells in Ada

Expert systems are best built using a *shell*, like ART (automated reasoning tool) or KEE (knowledge engineering environment). These shells have their own syntax and provide a disciplined and highly structured way of building expert systems. The shell provides not only the inference mechanism for the expert system but also the modular and hierarchical organization. This area provides the most promise for the use of Ada.

The shell structure can be used to limit the complexity of features used and their interaction. Further, the shell can generate a large number of type declaration or long select statements where this would be impossible by hand. This is often what is necessary to cope with strong typing.

In an expert system, general attribute propagation among rules requires garbage collection. However, the problems with the data abstractions in Ada can be dealt with if, for instance, explicit subprogram calls are inserted at key points in the Ada program to coordinate allocation and de-allocation. It is not feasible to have such calls inserted by a user, but they can be inserted reliably by the shell.

The modular and hierarchical aspects of shells are well supported by Ada. On a large system this will support team development well. However, as mentioned above it is necessary to have version control and recompilation analysis when using a compilable language. Languages such as Pascal or C would have very serious drawbacks in this environment. Fortunately, Ada is designed to support consistent separate compilation so it is very well suited to this task. However, during development the compilation costs could become very high.

3. Mixed Environments of Ada and AI Language

If Ada is only well suited to use with expert system shells, as described above, then other use of AI must use existing methods. Currently the accepted approach to dealing with the most general AI programs is the use of Common Lisp. Common Lisp is becoming the standard AI programming language in the U.S. Prolog is not yet a major force, although developments in this area should be watched, especially in light of Japanese efforts. The way to solve this dilemma is to integrate a Common Lisp system with the Ada environment. For proper integration such a Lisp system would need to be supplied by the same vendor that supplied the Ada system. In this way Ada can be integrated with AI Language tools and support. They can use shared list-processing and database packages and have the ability to call each other.

3.1 Impact on Development Tools

As long as Lisp components are under the same configuration management system, there should be no real problems. A Lisp system may require some of its own special tools, but these should not interact with the other tools.

3.2 Interfaces and Characteristics

The interface between Lisp and Ada is potentially complex. This can be made simpler by sharing standard packages (see below). However, in this case the only good solution is to require appropriate integration.

3.3 Operational Concepts

The biggest problem area in a mixed system is probably garbage collection. As described above, hand generated Ada is not designed to deal with this well. The only safe solution to this is to limit the actual AI work to the Lisp components. One can restrict the Ada components from allocation and de-allocation, unless they are correctly generated by, for instance, an expert system shell.

The storage management problem could be much simpler if it were possible to build packages that could deal with their own storage management without extra user calls. There appear to be only two ways to do this:

- low-level unsafe programming practices within the package
- language extensions to extend the data abstraction capabilities of Ada

Neither is particularly desirable.

3.4 Standard Packages

Standard packages that would be desirable for AI applications in Ada include:

- List Processing Predefined Package Library
- Associative Database Predefined Package Library

4. Conclusion

Because of its design goals Ada has some limitations in comparison with very powerful AI languages like Common Lisp. Except in very special applications, translation from Lisp to Ada is not feasible. Further, the modes of AI development are poorly supported by the Ada system. Ada is not well suited to the prototyping and incremental development required for AI work. Real promise, however comes in the area of expert system shells. The shells can be used to generate consistent Ada code that could not be generated by hand and further can generate complex constructs to bypass language feature mismatch problems. This should not be too surprising since the shell can use compiler implementation techniques used in Lisp.

If serious AI application beyond expert systems is anticipated, a mixed environment would be necessary. A language like Lisp provides the right language features along with support for AI style development. The most reasonable choice would be integrating Common Lisp in the Ada environment. However, access to some Lisp features from Ada would need to be restricted to ensure system reliability.

5. Bibliography

[1] Appelbe, W. F., "Abstract data types in Ada," Journal of Pascal, Ada and Modula-2, 3(1), 1984, pp. 26-29, 36.

[2] Crowe, M., D. Machay, M. Hughes, C. Nicol, "An interactive Ada compiler," Ada UK News, 6(4), Oct. 1985, pp. 47-50.

[3] Olgivie, J., "Using variant records: some basic Lisp functions in Modula-2," Journal of Pascal, Ada and Modula-2, 4(2), 1985, pp. 15-20.

[4] Poutanen, O., K-M. Varanki, T. Valimaki, "Notes on building a relational database management system in Ada," Ada in Use, Conf. Proc., Paris, May 1985, pp. 14-24.

[5] Schwartz, R.L. and P.M. Melliar-Smith, "On the suitability of Ada for artificial intelligence applications," Project 1019, July 1980, SRI International.

[6] Schwartz, R.L. and P.M. Melliar-Smith, "The finalization operation for abstract data types," Proc. of the 5th Int. Conf. on Software Engineering, March 1981, pp. 273-282.

[7] Stefik, M. and D. Bobrow, "Object oriented programming," AI Magazine, VI(4), 1986, pp. 40-62.

N89-16349

Intelligent User Interface Concept for Space Station

by:

Edward Comer, Cameron Donaldson, and Kathleen Gilroy
Software Productivity Solutions, Inc.

and

Elizabeth Bailey
Software Metrics, Inc.

Introduction

The space station computing system must interface with a wide variety of users, from highly skilled operations personnel to payload specialists from all over the world. The interface must accommodate a wide variety of operations from the space platform, ground control centers and from remote sites. As a result, there is a need for a robust, highly configurable and portable user interface that can accommodate the various space station missions.

This paper presents the concept of an intelligent user interface executive, written in Ada, that would support a number of advanced human interaction techniques, such as windowing, icons, color graphics, animation, and natural language processing. The user interface would provide intelligent interaction by understanding the various user roles, the operations and mission, the current state of the environment and the current working context of the users.

In addition, the intelligent user interface executive must be supported by a set of tools that would allow the executive to be easily configured and to allow rapid prototyping of proposed user dialogues. This capability would allow human engineering specialists acting in the role of dialogue authors to define and validate various user scenarios. The paper will discuss the set of tools required to support development of this intelligent human interface capability and will outline the prototyping and validation efforts required for development of the Space Station's user interface.

The Space Station User Interface Problem

The space station user interface represents one of the greatest challenges in human-machine interaction to date. The development, operation and use of the space station will involve thousands of people from all over the world. The space station user community will include private industry, universities, and other government agencies as well as the various NASA centers and their contractors.

The space station user interface must provide support for traditional ground-based, on-orbit and payload operations, each of which involves numerous operational roles. The test and integration function is representative of the diversity of these roles [DOR83]:

- o mission and operations planning
- o simulation and modeling
- o manufacturing development and test

- o pre- and post-launch integration and testing
- o on-orbit integration and testing
- o on-orbit maintenance and repair
- o payload integration and testing
- o user payload data processing
- o environment monitoring and control
- o real-time flight and operaitonal functions

While the need to support traditional operational roles like launch and flight control will continue with the space station, an increasing number of users will not have experience with NASA mission operations. As a multi-purpose facility, the space station will support users of its scientific laboratories and payloads, users running manufacturing and repair operations, and users providing transportation services.

Space station activities will be distributed over many sites (both government and commercial), including space platforms, maneuvering vehicles, ground-based command stations, and data collection centers. Many of the activities currently performed by ground-based personnel on specialized systems will have to be executed on the space platform using multi-purpose equipment.

Analysis of user interface technology currently in use on NASA projects demonstrates that it is clearly not adequate to meet the space station challenge. Some of the problems that must be addressed include:

- o Integration with other systems and off-the-shelf products (currently difficult or not possible)
- o Lack of support for advanced interaction techniques
- o Inadequate development tools
- o Lack of uniformity - Interfaces differ from system to system, payload to payload, and site to site
- o Difficult to use - require the assistance of specialists to accomplish mission (not applications-oriented); not tailorable to needs of individual users; poorly human-engineered;
- o Modifications often require reimplementation
- o Difficulty in performing validation in either off-line or real-time modes

Benefits to be derived from improving the current user interaction approach include:

- o Reduced life cycle cost by providing the necessary flexibility for users to accomplish new mission operations, and longer life of the operational software due to increased adaptability [BAS85]

- o Greater level of automated support, providing easier operations, use, modification, maintenance and validation [DOR85]

- o Increased operational confidence because personnel can perform activities themselves [DOR85]

Types of Interaction to be Supported

A variety of user-interaction styles have been made possible by advances in hardware technology. With an increase in the amount of the information that can be displayed and the operations available, the potential for effective and highly usable interfaces is greatly increased. Given the anticipated number of users and wide variety of user profiles for the Space Station computing system, it is essential that the user interface take advantage of proven sophisticated technologies such as advanced graphics, animation and natural language.

Graphics may be used in any of a number of ways to support the Space Station mission, including map generation, reading and analysis, decision support aids, teleconferencing, modeling and simulation, and the generation of forms, reports and presentations. This variety of applications places special requirements on the graphics functionality. Functional requirements can be separated into graphics output capabilities, graphics input capabilities, and the storage, retrieval and transfer of graphics information. An excellent detailed discussion of the classes of graphic interaction and techniques to support them is provided in [FOL84].

Output capabilities needed include support for display of chart, graph and other two-dimensional diagrams, display of image data, and support for high quality typography, a variety of colors in display output, and animation (discussed in later paragraphs). The use of color has evolved so rapidly that most systems do not effectively use it. Useful guidelines for the exploitation of color can be found in [MUR84]. It is expected that dynamic display capabilities, such as "zoom," "shrink," "pan," and "highlight" will be needed also.

Command and control applications typically require a significant amount of user interaction. Graphics capabilities must support interactive text entry, input of data in video, fax, or digital format, the development and management of menu-driven systems, the development and management of sophisticated multi-window applications, and screenpainting. The use of menus and windows is discussed in later paragraphs.

While pictures may be created, displayed and discarded "on-the-fly," it is often desirable to store pictures or portions of pictures for later use. Without a storage and retrieval capability, the reuse of a commonly needed picture (such as a map) would have to be accomplished by including the map-drawing program in every new application. A better method is for the application program to simply request that the map be retrieved from the common database and displayed. In addition to storage, the capability to transfer a picture from one system to another must be provided. Support must be provided for the generation and interpretation of pictures expressed according to a particular protocol.

A number of national and international graphics standards exist, the benefits of which are independence of application programs from device and vendor dependencies, thereby improving the portability of application programs and data. These standards represent a hierarchy of graphics software layers which can be compared to the Open Systems Interconnection (OSI) model for communications software. [HIN84] Among the most promising standards are the Graphical Kernel System (GKS) and Programmer's Hierarchical Interactive Graphics Standard (PHIGS). GKS functionality ranges from simple passive output to complex interactive graphics, and developments are underway to support three-dimensional graphics (GKS 3-D). ANSI language bindings, including Ada, are part of the GKS standard, and Ada implementations of GKS exist. [LEO85] PHIGS was designed to be upward compatible from GKS and an Ada binding for PHIGS has been developed.

Personal workstations with high resolution bit-mapped displays continue to decrease in cost, making available and very attractive the creation of animated drawings. Animation is a wonderful technique for illustrating dynamic objects and their actions. A very interesting discussion of the use of animation within a programming environment is provided by London and Duisberg. [LD85] The authors describe the need for an animation toolkit - a set of easily learned, easily applied, portable animation routines to relieve the tedious programming associated with computer animation. Such a toolset would include a library of reusable and connectable animation routines for creating new views. An animation toolset should be provided in the Space Station Development Environment along with packages for developing menu- and window-based systems.

Menu-driven systems have become commonplace for command and control applications. For the number of options and the sizes of databases we anticipate for the Space Station computing system, typical tree-structured menu systems will not be sufficient. The problem with these menus, which typically have numerous entries, is that they consume precious screen space and force the user to spend valuable time searching for a particular entry. Popular mechanisms to solve this problem include partitioning of entries according to logical function, pop-up submenus for related but more specific entries, paged menus and scrolling menus.

In any menu system, accomodation must be made for both naive and expert users, which implies that there must be an alternate route for commanding or selecting entries. Experience has shown that expert users will memorize such alternate routes to avoid using menus whenever possible. These alternate routes should be made obvious in training and documentation and designed to be consistent across the user interface. With an intelligent user interface which understands the user's role and experience level, continual customization and optimization of menus could be made (tailoring to the user profile). For example, reorganization of menu entries in accordance with frequency or infrequency of use may be in order. Existing menu systems used by NASA, such as the Transportable Applications Executive (TAE) do not include these flexible and intelligent capabilitites. [TAE85]

Multiwindow communication is desirable in situations where the user is concurrently performing many tasks. An example would be monitoring and analysis in a mission control center where displays are updated simultaneously by one or more real-time processors. Windowing capabilities are provided by a window manager, which both presents information in windows and allows the user to manipulate windows. Most window management systems fall into one of two categories: "tiling" or "desktop." Tiling involves arranging windows so that no overlap of windows occurs on the display screen (tiling is used in the Xerox Cedar System and the Microsoft Windows system). A desktop window manager does exactly the opposite - rectangular windows overlap like pieces of paper on a desk (the Smalltalk environment developed at Xerox PARC demonstrates a desktop window manager).

There are advantages and disadvantages to both types. Desktop window managers offer the user the most flexibility in arranging windows but at the same time require the user to perform an inordinate number of functions relating to the rearrangement of windows. The tiling model relieves the user of most of the window management functions but typically performs automatic resizing and rearranging which may not be suitable or desirable. Perhaps the best choice is a combination of the tiling and desktop schemes, where the desktop model is employed when the user is performing many different tasks at one time, and the tiling model is employed when the user is coordinating many views or actions to accomplish a single goal. It is essential also that the user be able to easily and quickly move, size, and cover windows, and be able to move information from one window to another (cut and paste operations). The proposed intelligent user interface could assist in the manipulation of windows by understanding the application domain and choosing sizes and arrangement as appropriate.

Some user interfaces employ the use of icons in conjunction with windows. Often, icons are used to symbolize available software utilities (such as mail) and document folders. Icons could be used to provide the user with valuable information regarding the context of his working environment. For example, for each window an icon could be provided which tracks the progress, associated files and problems with the window's associated task. Such information assists users who may otherwise lose track of what they are doing. This useful concept is illustrated in the PERQ Sapphire window manager. [SAP] Because of the space station computing system's projected international use, the use of icons may be helpful throughout the interface, although care must be taken not to use an icon which is culture-peculiar (e.g., a "mailbox" may not be very communicative outside of the U.S.).

Requirements for Intelligence in the User Interface

The diversity of users and missions for the space station presents a formidable challenge in the design of a generalized user interface executive. Current technologies in natural languages and expert systems point to numerous potential instances whereby the performance of the user interface could be significantly enhanced through the addition of intelligence.

One significant opportunity for improving user interfaces discussed previously (especially for naive users) is to incorporate natural language interfaces. Currently, we do a reasonably good job of literal interpretation of English sentences in static contexts and limited, well structured domains of application. [ITW83] Yet many of the natural language technologies can also be applied to intelligent command interpreters and query processors. As a result, significant benefits can be realized by both the inexperienced or casual users (e.g., payload telescience) and by highly skilled operations, test and integration personnel.

An intelligent user interface could translate loose or shortened queries or commands provided by the user into correct and fully qualified messages to the space station computing systems based on stored knowledge of:

- o Missions
- o Individual user roles within the missions
- o Operational environment configuration
- o Operational environment state
- o Individual user characteristics
- o User's current context

Stored knowledge of space station missions would define the underlying bases for communication by:

- o **Establishing the vocabulary**, including abbreviations, acronyms, synonyms, generalizations, set memberships, abstractions, type inhertances, etc. [BRA83]

- o **Defining acceptable actions**, that would include prepatory decisions, test actions, main goals, cautionary actions, concluding actions or enablement actions. [GAL84]

- o **Establishing thematic role frames**, that specify anticipated or allowable action themes involving the thematic object being queried or commanded, the agent for action, instruments involved in the action, along with action descriptors including source and destination, trajectory, location, time or duration. [WIN84]

This level of knowledge allows robust interpretation of queries or commands, whether provided by natural language input or by more structured language-based inputs. Additional capability can be added to the user interface if knowledge of individual user roles within the missions are also provided. This would allow the user interface to:

- o **Restrict user actions**, providing another level of security at the user interface.

- o **Forgive erroneous or flawed input**, now that the bounds of an individual's interaction is known.

- o **Provide more power** in the user interface by calling up scripts [SHA85] of frequent or allowable action sequences.

Knowledge of the environment configuration and the environment state would allow yet another level of user input checking. Because ground-based commanding of space station operations or payloads must bear the response delays of ground routing and satellite links, it is desirable to provide the maximum amount of user input checking at the point of input. Certainly, one would want to restrict any actions that are dangerous or detrimental to the platform, payloads or mission. While there would undoubtably be checks made at the point of commanding, an additional layer of user input fault tolerance is often necessary.

Knowledge and observation of user characteristics would allow static and dynamic tailoring of the user interface for individual users. The most obvious application would be in acknowledging or inferring the skill levels of users and modifying the user interface input and presentation modes accordingly. This would circumvent the frequent problems associated with powerful interfaces being hard to learn and "user-friendly" interfaces getting in the way of experienced users. In addition, the user interface could employ selective dissemination of information techniques which can dynamically tailor the method presentation (e.g., text vs. graphics) to the user profile [ITW83] or selective omission of information techniques (e.g., abstraction, indexes, summarization) [WIN84].

The technologies exist today to provide the capabilities described above in a cost-effective, low risk and timely fashion for space station. Over time, the completeness and robustness of the knowledge base would improve, providing an increasingly powerful user interface.

In the long term, knowledge of the users context will provide the most significant improvement in the user interface. Here, the intelligent user interface would attempt to understand the users intent and recognize the plan being pursued. In this mode, the user interface would constantly be analyzing a user's interaction in light of his role and in light of the state of the environment to answer the following questions: [SHA85]

o Why is this character doing what he is doing?
o What are is motivations?
o What are his plans?
o What's his intention?

To accomplish the understanding of a user's context, additional research is needed in concept modeling and reasoning about goals and actions of rational agents (i.e., the user). [ITW83]. Once achieved, the user interface could become an active element of the user-computer dialog, instead of a passive one. This could be most important when responding to emergency or abnormal circumstances where the user must quickly take some form of alternate action. In such a situation, the user does not have the time (or often the presence of mind) to completely describe a new course of action. Instead, he would leverage the machines knowledge of context and succinctly execute a new set of actions, such as: [HAM84]

o Use alternate agent to accomplish the goal
o Use alternate plan
o Execute script rapidly
o Wait out current state
o Jump into the middle of the script
o Counterplan against a potential future state
o Put up with it
o Recover

In addition to augmenting the operational user interface with knowledge-based capabilities, there is also significant potential for assisting the dialog design, prototyping and management tasks with expert system capabilities. Mission and user role knowledge can assist in simulating user interfaces and rapid prototyping. An expert system could assist the dialog authors in selecting interaction approaches and configuring the user interface accordingly. Similarly, there is a potential for assisting in the interpretation of user interaction data and metrics and in suggesting improvements or explaining perceived behavior. [ITW83]

Although there has been only limited work in incorporating artificial intelligence technologies into intelligent user interfaces, we are convinced that there is significant potential, particularly for a program as complex as space station.

User Interaction Design and Validation

A real challenge lies in combining these user-interaction capabilities in a way that is consistent and conherent for the particular users, their tasks, and their environment. This is an especially difficult task because capabilities or features which may be desirable for one class of users, type of activity or operational environment may not be for another. For example, features which support ease of learning are needed for the inexperienced or infrequent user while features to enhance efficiency and power are likely to be far more important for experienced or every-day users. The characteristics of the operations or tasks that are carried out have implications for user-interface design as well. For tasks that are repetitive in nature (e.g., data entry and text editing) efficiency of physical actions (such as number of key strokes) is important. For tasks requiring a high level of mental effort, minimizing the user's mental load and reducing errors is more important.

In short, the features required to support effective user interaction can mean different things for different users and types of tasks. In addition, users themselves are not static entities. Ideally, one would like user interfaces which evolve as a given user gains experience and sophistication, both with the task and with the computer system.

The major components of effective user-interface design include:

- the ability to evaluate key features of user interfaces, especially at an early point in the development
- the availability of a toolset to support user-interface development
- a system architecture which allows development of user interfaces to proceed independently and in parallel with development of the rest of the system.

Evaluating User Interfaces

The design of user interfaces should proceed in a much more iterative fashion than the design of other parts of the software. There are too many unknowns concerning which combination of user-interface capabilities will best suit the various types of users, their tasks and operational environments. At the same time, there are few design principles to which a developer can turn for concrete guidance. Even obviously important principles such as "consistency" can be difficult to apply in practice since the designer's concept of consistency may not fit the users. Design decisions which seem obvious to the developers can lead to confusion among users.

User behavior can be a valuable source of guidance in selecting user-interaction capabilities. By observing how users accomplish a given task, what errors they make, how much time they require, and so on, the designer has an objective and meaningful basis for choosing among alternatives and for confirming the usability of choices already made. The earlier one can begin to gather this type of information the better, using prototypes and simulations to test out design alternatives.

These evaluations can range from informal observational studies to formal standardized experiments. If the evaluation is concerned solely with identifying the strengths and weaknesses of a single design, then an informal observational study is sufficient. If the purpose of the evaluation is to compare alternative designs, then one must turn to the methodology of controlled experimentation, using a standard set of procedures in order to produce as unbiased an evaluation as possible. In either case, in order for the evaluation to be valid, the users, their tasks, and surrounding conditions must be representative of those that will be supported by the operational system.

Simulations may be of special interest in the design of the space station because they can be used to evaluate user-interaction capabilities that do not yet exist, thus providing information about the likely benefits resulting from various technologies that may require substantial resources to implement. Gould, Conti, and Hovanyecz [1] carried out this type of study by simulating a "listening typewriter" that could take human speech as input and produce a printed version of that speech as output. A human typist hidden from view simulated the speech recognition capabilities required for the typewriter.

Tools for Developing User Interfaces

In light of the above discussion, facilities are needed for recording user interaction. The level of detail of the information captured can vary from an atomic level (e.g., every keystroke) to a much higher level such as the total time required to complete a given task or a summary of the different commands used. The level of detail will obviously depend on the question of interest. In evaluating a text editor, for example, one may wish to log a time-stamped record of all keystrokes. Tools are also needed for prototyping user interfaces not only in terms of static displays but in terms of the dynamic aspects of an interaction as well.

Tools and associated databases are needed to assist in defining user-input languages and in creating, editing, and storing displays and display definitions of all types including graphics, text, animation, menus, and forms. The Dialogue Management System (DMS), developed by Hartson and his colleagues [3], contains many of these capabilities.

Architecture for the Intelligent User Interface System

Given the iterative nature of user-interface design, one of the key properties desired of user interfaces is flexibility. The space station computing system must allow changes in user interfaces as a result of improvements suggested by user testing or the addition of new users, new operations, or new operational sites. These changes must be made easily, quickly, and without adversely impacting other parts of the software.

Software designers have traditionally isolated the software from the effects of hardware changes. In the same way, the computational or functional portion of the software should be isolated from changes in the portions controlling the user interface. Hartson and his colleagues [3] have written extensively about the architectural issues involved.

Communicating with the user, including all input checking, should be the responsibility of the user-interaction components while the correct and efficient functioning of the system functionality should be the responsibility of the computational components. Hartson has argued for a parallel separation in the skills required to design, implement, and test these two components with the user interface falling within the domain of the human-factors specialist and the computational portion belonging to the traditional software designer and programmer. Once the interface has been defined between these two components, the two types of specialists can work independently and in parallel without interference.

Because of the complexity of such a user interface and the complexity of the various missions or roles, it is necessary to develop a support system for the user interface. We propose a comprehensive user interface system consisting of the following:

o **User Interface Executive.** This Ada software package would provide the user interface utilities embedded within the operational space station computing systems and be configured for the specific machine and missions for each installation via a resident database and knowledge base.

o **User Interface Prototyping Subsystem.** This would bundle the user interface executive with generalized simulation capabilities and data monitoring and collection routines. This subsystem would provide a pnambic ("pay no attention to the man behind the curtains") laboratory for user interface experimentation.

o **User Interface Configurator.** This software would customize the user interface executives for installation and also the prototyping subsystem for experimentation. The customization provided by this tool would include input and presentation options and the higher level dialog customizations. The low level customizations would be accomplished through a combination of program directed software builds from a library of user interface primitives and parametric or language-driven initializations. Higher level customizations will be accomplished through a rule compiler that will configure the user interface executive with the required knowledge and inference algorithms.

o **Dialog Management Subsystem.** This subsystem would input knowledge regarding the mission, users, configuration, etc., and be used to compose and configure dialog sessions. Data received from the user interface prototyping subsystem would be analyzed to validate dialog session before deployment.

The user interface system would be designed with an open architecture to allow easy expansion as new user interface technologies become available. The approach discussed in this paper will naturally put greater requirements on the local processing capabilities of the user interface devices. Current declining cost trends in high resolution graphic workstations leads us to believe that the increased functionality received from an intelligent user interface will be a cost effective solution for the space station. In addition, the proposed open architecture will lend itself to additions of new technologies over the space station life cycle.

References

[BAS85] Basili, V. "A View of Language Issues", presented at Open Forum on Space Station Software Issues, Apr. 25, 1985.

[BOE85] Boehm, B. "A View of Software Development Environment (SDE) Issues", presented at Open Forum on Space Station Software Issues, Apr. 24, 1985.

[BRA83] Brachman, Ronald J. ,"What IS-A Is and Isn't: An Analysis of Taxonomic Links in a Semantic Network," Computer, IEEE Computer Society, Vol. 16 No. 10, October 1983.

[DOR83] Dorofee, A. and Dickison, L. "High Order Language: Second Level White Paper, Space Station Operations Working Group, KSC/DL-DED-22, Jul. 29, 1983.

[DOR85] Dorofee, A. and Dickison, L. "Space Station Operations Language System Requirements and Concept Definition (Preliminary)", KSC, Aug. 1, 1985.

[GAL84] Galambos, James and Black, John, Using Knowledge of Activities to Understand and Answer Questions, Yale University Cognitive Science Program, Cognitive Science Technical Report #29, August 1984.

[GCH82] Gould, J.D., Conti, J., and Hovanyecz, T. "Composing Letters with a Simulated Listening Typewriter". In Proceedings of the Human Factors in Computing Systems Conference, ACM, Washington, D.C., 1982.

[HAM84] Hammond, Kristian, Indexing and Causality: The Organization of Plans and Strategies in Memory, Yale University Department of Computer Science, YALEU/CSD/RR#351, Dec 1984.

[HJ85] Hartson, H.R. and Johnson, D.H. "Dialogue Management: New Concepts in Human-Computer Interface Development". ACM Computing Surveys, 1985.

[ITW83] Report of the Information Technology Workshop, National Science Foundation, AD-A144-212, 1 Oct 1983.

[SHA85] Shank, Roger C., Questions and Thought, Yale University Department of Computer Science, YALEU/CSD/RR#385, August 1985.

[SOW84] Sowa, John F., Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley Publishing Company, 1984.

[WIN84] Winston, P. H., Artificial Intelligence, Addison-Wesley Publishing Company, 1984.

[YH86] Yunten, T. and Hartson, H.R. "A SUPERvisory Methodology Notation (SUPERMAN) for Human-Computer System Development". In H.R. Hartson (Ed.), Advances in Human-Computer Interaction, Ablex Publishing Corporation, 1986.

[HIN84] Hinden, H. "Graphics Standards Finally Start to Sort Themselves Out," Computer Design, May 1984, pp. 167-180.

[LEO85] Leonard, T. "Ada and the Graphical Kernel System," Ada in Use: Proceedings of the Ada International Conference, May 1985, pp. 136-150.

[LEO85] Leonard, T. "Ada and the Graphical Kernel System," _Ada in Use: Proceedings of the Ada International Conference_, May 1985, pp. 136-150.

[GAR84a] Garman, J. "Data Processing for Space Station", Oct. 18, 1984 (presentation)

[GAR84b] Garman, J. "Networking and Data Processing for the Johnson Space Center", Dec. 13, 1984. (presentation)

[HAL85] Hall, D. "Space Station and the Role of Software", presented at Open Forum on Space Staton Software Issues, Apr. 24, 1985.

[SAP] Myers, Brad A. "The User Interface for Sapphire," _IEEE Computer Graphics and Applications_, Volume 4, Number 12, December 1984.

N89-16350

AN ADA INFERENCE ENGINE FOR EXPERT SYSTEMS

David B. LaVallee
Ford Aerospace and Communications Corp.
College Park, Maryland

1 INTRODUCTION

The purpose of this research project is to investigate the feasibility of using Ada for rule-based expert systems with real-time performance requirements. This includes exploring the Ada features which give improved performance to expert systems as well as optimizing the tradeoffs or workarounds that the use of Ada may require. A prototype inference engine for general purpose expert system use was built using Ada, and rule firing rates in excess of 500 per second were demonstrated on a single MC68000 processor.

The knowledge base uses a directed acyclic graph to represent production rules. The graph allows the use of AND, OR, and NOT logical operators. The inference engine uses a combination of both forward and backward chaining in order to reach goals as quickly as possible. Future efforts will include additional investigation of multiprocessing to improve performance and creating a user interface allowing rule input in an Ada-like syntax.

Some of the issues discussed concerning Ada's use in expert systems include: How should a knowledge base be structured in Ada? How should the knowledge base be searched, especially in the context of a dynamic problem space with new data constantly entering the system? Can real-time performance be achieved?

A critical issue involves the use of Ada's multitasking to implement parallel algorithms in expert systems. Clearly the inference engine can be implemented as a single task which can be integrated into a larger system and execute only when necessary. However, the execution of the inference mechanism in a parallel manner should increase performance. Using segmented knowledge bases, backward chaining in parallel on all goals at once, and forward chaining in parallel on individual rules are some of the different strategies to be considered. These strategies use different levels of granularity. Using an algorithm with a low level of granularity, fewer parallel computations will be performed and intertask communication will be less frequent. Using a high level of granularity, much computation is done in parallel, however it involves considerable intertask communication. The overhead involved in creating tasks and in communicating between them, must be weighed against the benefits of the parallel performance.

2 EXPERT SYSTEM USE IN THE SPACE STATION

The Space Station will be a tremendously complex system. The automation of many of the Space Station activities and related monitoring functions in a safe and reliable manner will help to increase the efficiency and cost effectiveness of the system. In addition, one of the key engineering guidelines for the Space Station is that it should be able to carry out normal operations for some finite period of time without contact with the ground. As pointed out in a NASA Technical Memorandum on Automation Technology For The Space Station [1],

> "Expert systems are needed to perform many monitoring and control functions requiring complex status analysis and automated decision making so that the Station is less dependent on ground support in these areas."

Also in [1],

> "In emergency situations, automated systems which respond very rapidly to a crisis can bring the system to a fail-safe condition before extensive damage occurs... Without automation, humans may be placed more often in pressure-prone situations such as EVA and emergency maintenance in which there is an increased chance of error."

Expert systems could incorporate fault diagnosis, isolation, and recovery to enhance crew safety. Alarms could be triggered automatically to warn crew members of hazardous situations. In addition, many faults could be corrected before they pose any danger to the crew or spacecraft.

3 FORD ADA INFERENCE ENGINE

3.1 Description

The Ford Ada Inference Engine (FAIE) is a research prototype expert system inference engine designed to execute as an Ada task embedded in an expert system which could in turn be embedded in a larger program. The sample application discussed here involves using FAIE for fault diagnosis. A typical rule in this type of system might be:

"IF temperature is above normal and
heater output is above normal,
THEN power off heater."

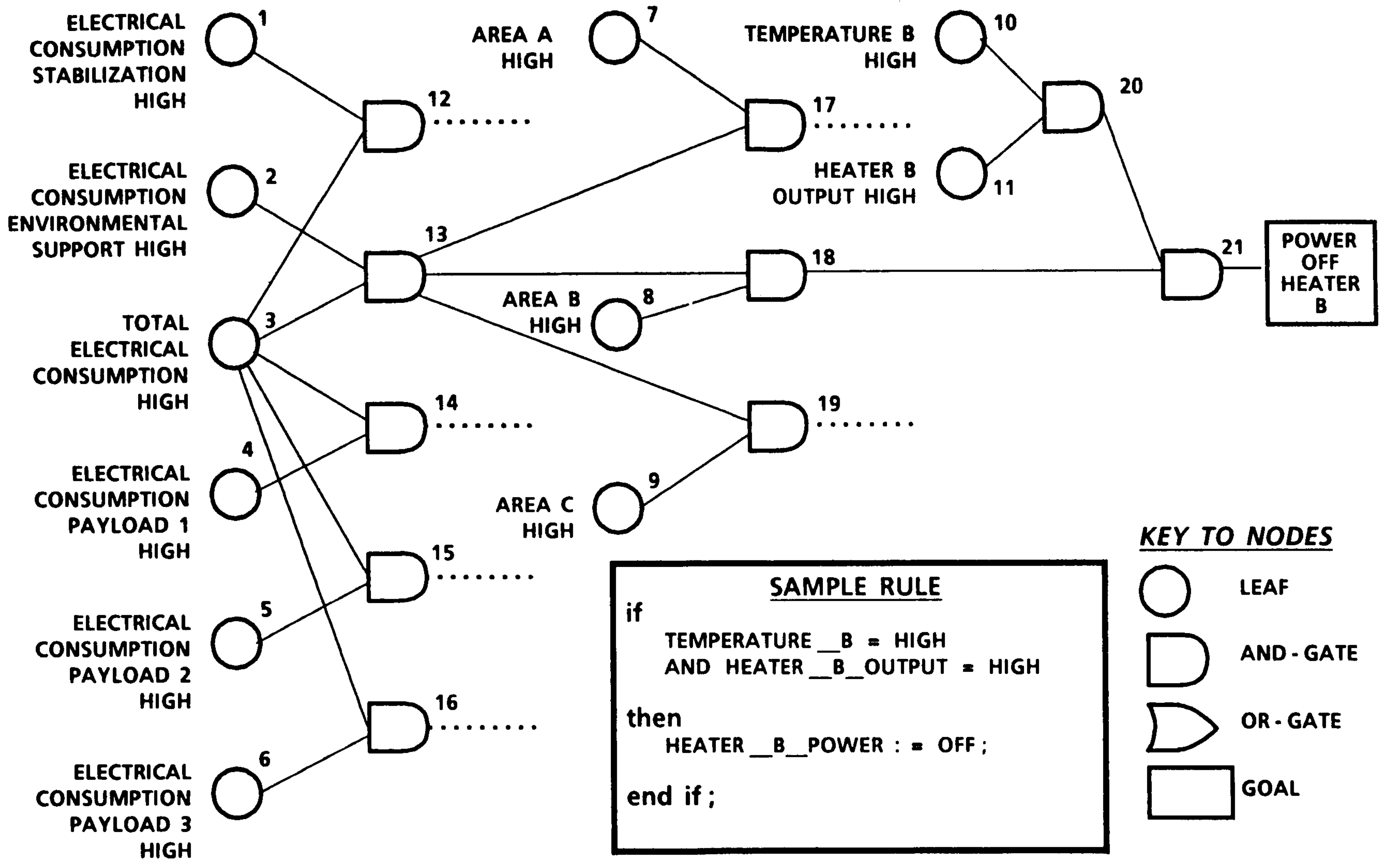
The knowledge base is structured as a directed acyclic graph. This can be thought of as a network of nodes with the links all pointing in the same direction. For the diagnostic system, the leaf nodes on one side of the graph represent the various sensor data measurements. Commands for corrective action are the goal nodes on the other side of the graph. The relationships between erroneous measurements are the intermediate nodes leading to a goal. Figure 1 shows a portion of a sample graph. Note: the dotted lines represent additional portions of the graph that are not shown.

The leaf nodes represent initial data points that must be provided to the inference engine. The nodes on the other side of the graph represent goal states that are sought when executing the inference engine. The nodes in between represent hypotheses or subgoals that will be tested. The links between the nodes are the "production rules" that the inference engine uses to traverse the graph.

Since we have a compiled, static knowledge base, all elements are present in the graph. Each node has a status which we will refer to as "flagged", "unflagged", or unknown. A "flagged" node is one that satisfies its associated IF-THEN rule. We must distinguish between an untested node (status equals unknown), and a node that was tested and does not satisfy the associated IF-THEN rule (status equals "unflagged"). A "flagged" node is one that will be used to traverse the graph. The path to a goal must be continuous through "flagged" nodes. An "unflagged" node represents a "dead end".

Status for all the leaf nodes is passed to the inference engine when a problem exists. Figure 2 shows the sample knowledge base with all the leaves (nodes 1-11) given an initial status. Nodes 2,3,10 and 11 are "flagged".

In an attempt to find a goal as quickly as possible, the successors of the first "flagged" leaf node are examined and the first one in the list is visited using Ada procedure FORWARD_CHAIN. Since the status of the successor node is initialized to unknown, its predecessors are examined along with its AND/OR flag to determine its status. If the status of this first successor to the first leaf node is found to be "flagged", then its first successor in its list is visited, and so on until a goal is found or a dead end is reached. If the status of this first successor is found to be "unflagged", then the next successor in the first leaf node's list is visited.


ELECTRICAL CONSUMPTION STABILIZATION HIGH
1
ELECTRICAL CONSUMPTION ENVIRONMENTAL SUPPORT HIGH
2
TOTAL ELECTRICAL CONSUMPTION HIGH
3
ELECTRICAL CONSUMPTION PAYLOAD 1 HIGH
4
ELECTRICAL CONSUMPTION PAYLOAD 2 HIGH
5
ELECTRICAL CONSUMPTION PAYLOAD 3 HIGH
6
AREA A HIGH
7
AREA B HIGH
8
AREA C HIGH
9
TEMPERATURE B HIGH
10
HEATER B OUTPUT HIGH
11
12
13
14
15
16
17
18
19
20
21
POWER OFF HEATER B
KEY TO NODES
LEAF
AND - GATE
OR - GATE
GOAL

SAMPLE RULE

```
if
    TEMPERATURE__B = HIGH
    AND HEATER__B__OUTPUT = HIGH

then
    HEATER__B__POWER : = OFF ;

end if ;
```

Figure 1. Sample Compiled Knowledge Base

If the status of a predecessor node is unknown, then Ada function BACK_TRACK is invoked to return the status. Both subprograms FORWARD_CHAIN and BACK_TRACK are recursive.

Figure 3 shows the resulting status after running the inference engine. To get to Figure 3 from Figure 2 the following steps were taken:

1. Node 2's successor list is examined, and node 13 is passed in a call to FORWARD_CHAIN.
2. Since node 13 is an "and gate" and both its predecessors (2 and 3) are "flagged", node 13 becomes "flagged".
3. Node 13's successor list is examined, and node 17 is passed in a recursive call to FORWARD_CHAIN.
4. Since node 17 is an "and gate" and node 7 is "unflagged" node 17 becomes "unflagged".
5. FORWARD_CHAIN returns to visiting node 13, where the successor list is examined, and node 18 is passed in another recursive call to FORWARD_CHAIN.
6. Since node 18 is an "and gate" and both its predecessors (8 and 13) are "flagged", node 18 becomes "flagged".
7. Node 18's successor list is examined, and node 21 is passed in another recursive call to FORWARD_CHAIN.
8. Since the status of node 20 is unknown, node 20 is passed in a call to BACK_TRACK.
9. Since node 20 is an "and gate" and both its predecessors (10 and 11) are "flagged", node 20 is "flagged" and BACK_TRACK returns.
10. Since node 21 is an "and gate" and both its predecessors (18 and 20) are "flagged", node 21 is "flagged" and a goal has been found.
11. The recursive calls return and visit other successor nodes for additional goals.

3.2 Performance

The search speed is dependent upon the depth of the graph from leaf to goal but is independent of the number of leaves or goals in the graph. The only rules that are attempted to be matched already have at least one element of its left-hand-side "flagged". When a goal node is "flagged", the inference engine will issue a procedure call or task rendezvous to invoke logic associated with the goal state (e.g. turn a circuit on or off).

Neither heuristic pruning nor optimal search techniques are employed. Some control over program execution can be accomplished by ordering the leaf nodes and/or ordering the list of successors and predecessors. Factors such as severity of problem or frequency of occurrence can be used to prioritize these lists.

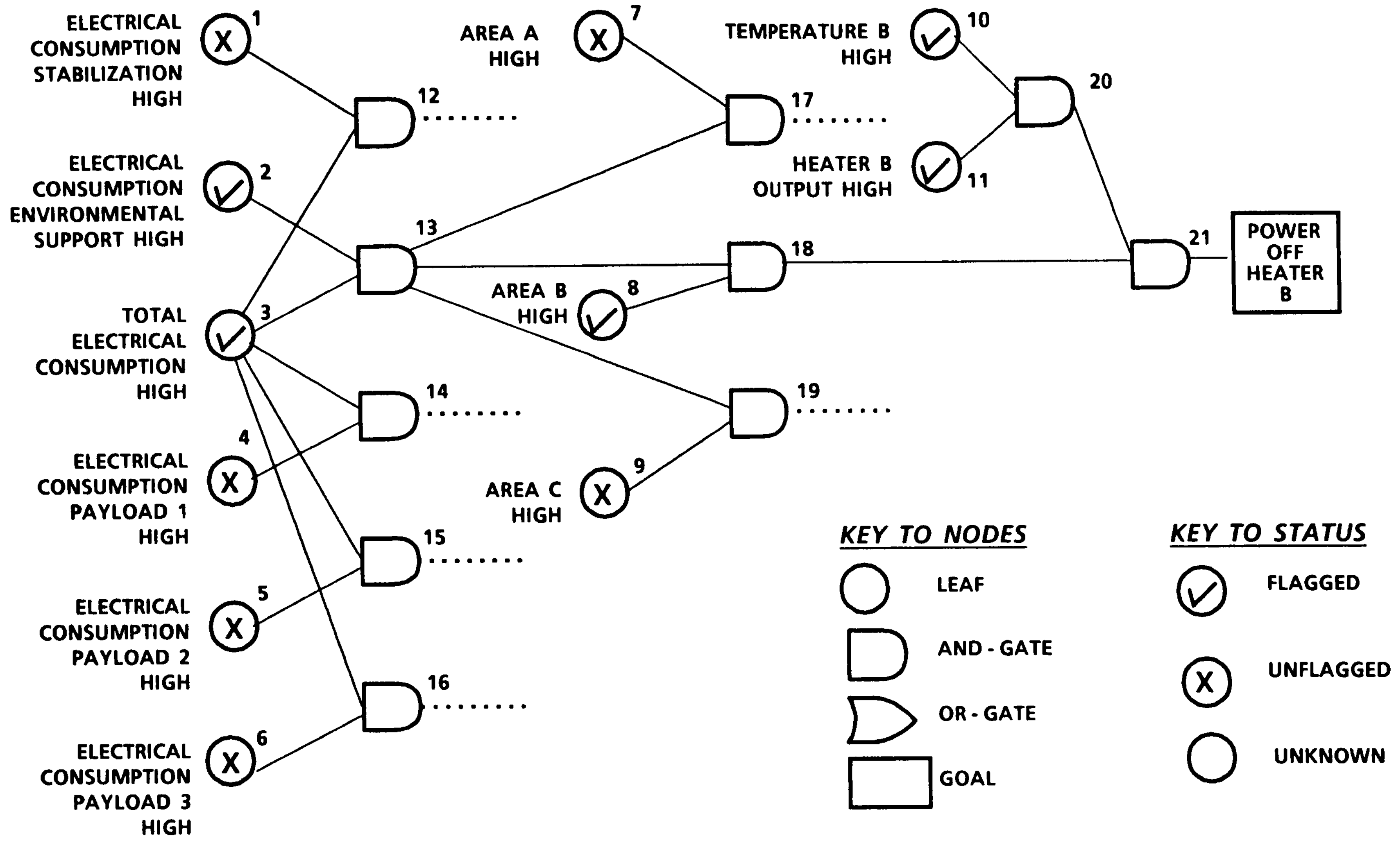

ELECTRICAL CONSUMPTION STABILIZATION HIGH
1
ELECTRICAL CONSUMPTION ENVIRONMENTAL SUPPORT HIGH
2
TOTAL ELECTRICAL CONSUMPTION HIGH
3
ELECTRICAL CONSUMPTION PAYLOAD 1 HIGH
4
ELECTRICAL CONSUMPTION PAYLOAD 2 HIGH
5
ELECTRICAL CONSUMPTION PAYLOAD 3 HIGH
6
12
13
14
15
16
AREA A HIGH
7
AREA B HIGH
8
AREA C HIGH
9
17
18
19
TEMPERATURE B HIGH
10
HEATER B OUTPUT HIGH
11
20
21
POWER OFF HEATER B
KEY TO NODES
LEAF
AND - GATE
OR - GATE
GOAL
KEY TO STATUS
FLAGGED
UNFLAGGED
UNKNOWN

Figure 2. Knowledge Base – Initial Problem State

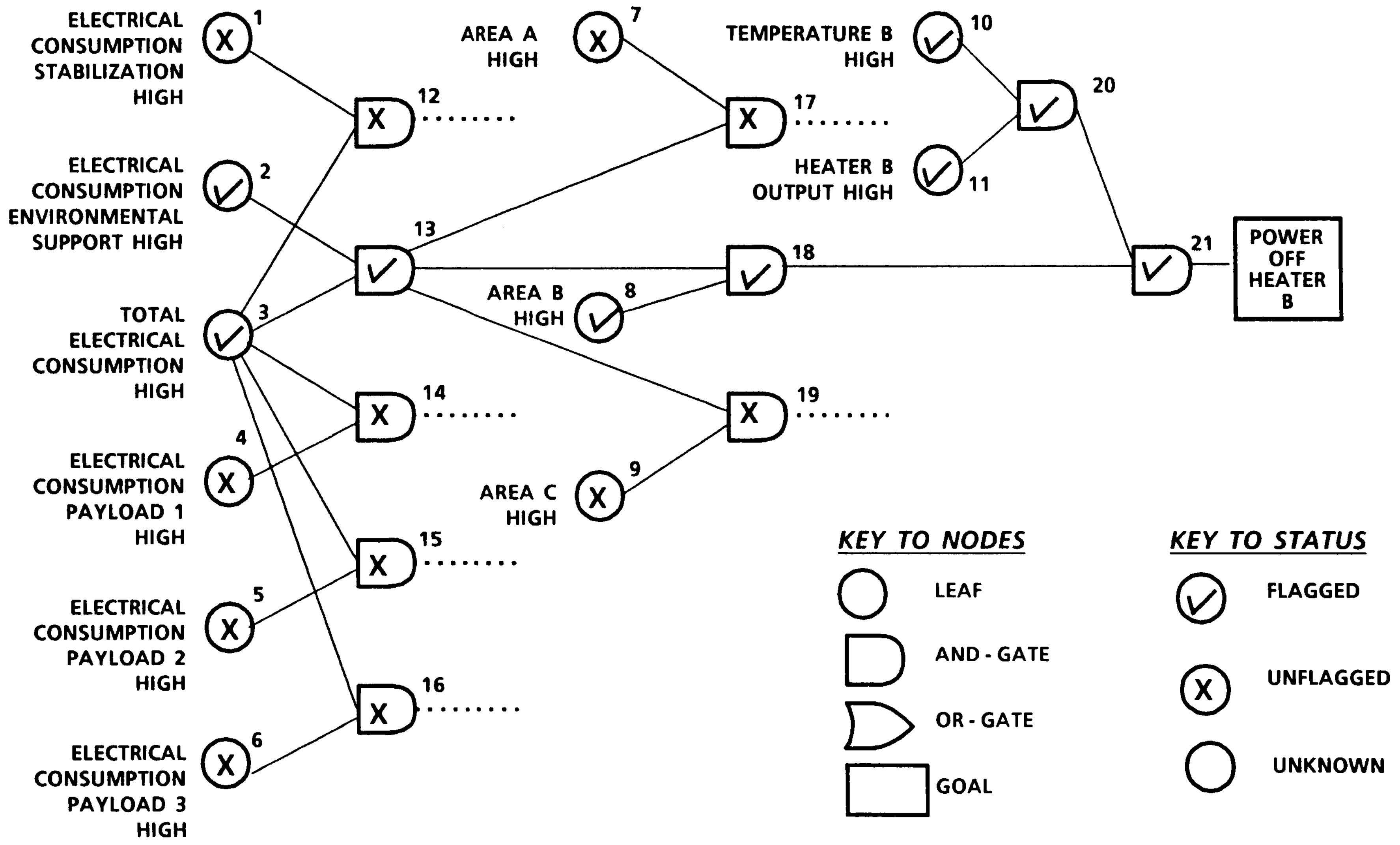

ELECTRICAL CONSUMPTION STABILIZATION HIGH
1
ELECTRICAL CONSUMPTION ENVIRONMENTAL SUPPORT HIGH
2
TOTAL ELECTRICAL CONSUMPTION HIGH
3
ELECTRICAL CONSUMPTION PAYLOAD 1 HIGH
4
ELECTRICAL CONSUMPTION PAYLOAD 2 HIGH
5
ELECTRICAL CONSUMPTION PAYLOAD 3 HIGH
6
12
13
14
15
16
AREA A HIGH
7
AREA B HIGH
8
AREA C HIGH
9
17
18
19
TEMPERATURE B HIGH
10
HEATER B OUTPUT HIGH
11
20
21
POWER OFF HEATER B
KEY TO NODES
LEAF
AND - GATE
OR - GATE
GOAL
KEY TO STATUS
FLAGGED
UNFLAGGED
UNKNOWN

Figure 3. Knowledge Base – Problem Solution

This design assumes that all calculations on the data are performed up front, prior to invoking the inference engine. Speeds in excess of 500 rule firings per second were executed on a single processor. A rule firing is defined to be "flagging" a node, increasing working memory. This is similar to results obtained by other non-LISP inference engines (e.g. OPS83 or the BLISS version of OPS5). These results indicate that real-time performance is achievable.

4 USE OF ADA FEATURES

The knowledge base is an array of records. Each record is a node with the following information:

STATUS - UNKNOWN, FLAGGED or UNFLAGGED
FORM - LEAF, SUBGOAL or GOAL
AND_OR_FLAG - AND or OR
POINTER TO PREDECESSOR LIST
POINTER TO SUCCESSOR LIST
TEXT STRING IDENTIFIER

The Ada package describing the data types in the knowledge base is given in Figure 4. A description of Ada constructs used to transform LISP research prototype expert systems into Ada production systems was given by Rude [2]. Unlike Rude, I have implemented the predecessor and successor lists as linked lists of records using access types rather than arrays of records. This allows flexibility in dynamically altering the knowledge base at runtime, e.g. if a sensor is determined to be faulty and you wish to ignore its input. In addition, the minimum amount of storage space is used. Using arrays would require that all nodes allocate space for the largest list of predecessors or successors and would also require re-compilation to adjust the maximum sizes.

Ada tasking was used to embed the expert system in a larger Ada program. It can stand idle while other monitoring and limit checking functions are performed and then spring into action when an anomaly is detected. A more extensive use of tasking can be made to perform various functions of the expert system in parallel. This will be discussed in the next section.

Although Ada provided adequate constructs to build this inference engine there are a couple of features of other languages (notably LISP languages) that would be very useful for expert systems if supported in Ada. The main feature desired is the ability to pass Ada functions as parameters in subprogram calls. An alternative would be the ability to embed a function in a data structure, such as the field of a record, to be executed when accessed. This could be used to perform calculations when needed. As mentioned earlier, in this version, all calculations needed to execute the inference engine must be performed up front.

The second desired feature is the ability of an object to inherit values from a parent. For example, when new elements are added to a linked list or tree-like structure, they could inherit values in specified fields of their parents. This would reduce subprogram calls and a number of extra objects for data storage.

5 FUTURE INVESTIGATION

5.1 Further Multitasking Work

One main thrust of our further work will focus on the use of multitasking to improve performance. This will also solve the problem of reading dynamic data which is constantly being updated as inferencing is in progress. It seems reasonable to use Ada tasking to enhance the real-time performance of inference engines. Although true production-quality multiprocessing Ada compilers do not yet exist, it is now feasible to write tasking implementations of inference engines which will exhibit order-of-magnitude improvements in rule-firing rates when ported to true multiprocessing Ada environments.

Douglass [3,4] lists five levels of potential parallelism in rule-based expert systems. They are: subrule level, rule level, search level, language level, and system level. These levels include different types within them. Douglass concentrates on rule level and various types of search level parallelism. He gives a range of quantitative results for these levels using mathematical models and concludes that combinations of subrule, rule and search level parallelism will yield better results than any single level when the characteristics of the specific system are taken into consideration. He also mentions that very little work has been implemented and tested on parallel computers.

Communication between processes is an important factor in the efficiency of parallel algorithms. Generally speaking, the more frequently that information is exchanged, the slower the computation is performed since processes spend a larger portion of their time communicating rather than computing. Researchers working on the DADO machine [5,6] have developed some unique methods of communicating between parallel processors (e.g. a binary tree structure of processors with communication rules controlled by hierarchy).

In Ada, the task is the natural construct for parallel processing. However, multitasking involves considerable overhead in creating/activating tasks, communicating between them, and terminating them. This overhead must be compared with the amount of computation performed in parallel in order to determine the relative efficiency gained by various strategies of parallel processing. Gehani [7] concurs, and goes an to say that in designing concurrent programs in Ada, one must avoid the polling bias in the communication mechanism. He also points out that multiprocessing programs will be more efficient if the underlying hardware offers genuine concurrency.

Deering [8] also emphasizes that hardware considerations, especially processor speeds versus memory speeds, must be examined when designing the architecture of expert systems. He says one should "study hardware technology to determine at what grain sizes parallelism is feasible and then figure out how to make [the] compilers decompose programs into the appropriate-size pieces."

Granularity is the average amount of work done by a process between communication with other processes. It is inversely proportional to the frequency of communication. The five levels of parallelism mentioned by Douglass range from very finely grained to roughly grained. A fine grained approach was taken by Rude [2] where each rule was itself declared as an Ada task with rendezvous for links to predecessors and successors. This concept has merit but is questionable for real-time applications. In the implementation of the PICON expert system for real-time process control [9,10], a roughly grained algorithm was chosen by segmenting parts of the knowledge base and applying priorities to searching the different portions. Our future investigations will include analyzing various strategies, including forward and backward chaining on individual rules in parallel, dividing the knowledge base, and combinations of the different strategies.

5.2 User Interface

Another area for future work involves building a user interface for accurate and efficient knowledge acquisition. The accumulation of the domain knowledge and its insertion into a knowledge base has often been a bottleneck in expert system production. The Ada language IF-THEN-ELSE constructs are readable and English-like. We will build a user interface in an Ada syntax that is hopefully both easy for the knowledge engineer to use, and also easily translates into Ada code.

6 CONCLUSION

The prototype demonstrates the feasibility of using Ada for expert systems on a small scale. Investigation of multitasking and alternate knowledge base representations will help to analyze some of the performance issues as they relate to larger programs.

References:

1. NASA Advanced Technical Advisory Committee, Advancing Automation and Robotics Technology for the NASA Space Station and for the U.S. Economy, NASA Technical Memorandum 87566, Volume II, March 1985 p. 5.

2. Rude, A., "Translating a Research LISP Prototype to a Formal Ada Design Prototype", Proc. Washington Ada Symposium, March 1985.

3. Douglass, R., "Characterizing the Parallelism in Rule-Based Expert Systems", Proc. Hawaii International Conference on Systems Science, HICSS-18, Jan. 1985.

4. Douglass, R., "A Qualitative Assessment of Parallelism in Expert Systems", IEEE Software, May 1985, pp. 70-81.

5. Stolfo, S., and D. Miranker, "DADO: A Parallel Processor for Expert Systems", Proc. 1984 Int. Conf. on Parallel Processing, IEEE Computer Society Press, August, 1984.

6. Stolfo, S., "Five Parallel Algorithms for Production System Execution on the DADO Machine", Proc. of the NCAI, Austin, TX, 1984.

7. Gehani, N., Ada: Concurrent Programming, Prentice-Hall Inc., 1984.

8. Deering, M. "Architectures for AI", Byte Magazine, April, 1985.

9. Moore, R., L. Hawkinson, C. Knickerbocker, L. Churchman, "A Real-Time Expert System for Process Control", 1st Conf. on AI Applications, IEEE Computer Society Press, Dec. 1984.

10. Moore, R., "Adding Real-Time Expert System Capabilities to Large Distributed Control Systems", Control Engineering, April 1985.

```
with DYNAMIC_STRING;
package GRAPHS is
  type NODE_NUM is new INTEGER range 0..INTEGER'LAST;

  type STATUSES is (FLAGGED, UNFLAGGED, UNKNOWN);
  type GATE is (AND_GATE, OR_GATE);
  type NODE_FORM is (GOAL, SUBGOAL, LEAF);

  type PRED_NODE;          -- DATA STRUCTURE FOR LINKED LIST
                           -- OF PREDECESSORS
  type PRED_NODE_PTR is ACCESS PRED_NODE;
  type PRED_NODE is record
    NAME : NODE_NUM;
    NEG_LOGIC_FLAG : BOOLEAN := FALSE;
               -- FALSE = want pred to be flagged.
               -- TRUE = want pred to be unflagged.
    NEXT : PRED_NODE_PTR;
  end record;

  type SUCC_NODE;          -- DATA STRUCTURE FOR LINKED LIST
                           -- OF SUCCESSORS
  type SUCC_NODE_PTR is ACCESS SUCC_NODE;
  type SUCC_NODE is record
    NAME : NODE_NUM;
    NEXT : SUCC_NODE_PTR;
  end record;

  type NODE is record      -- DATA STRUCTURE FOR A NODE OF
                                        --    THE GRAPH
    STATUS    : STATUSES := UNKNOWN;
    AND_OR    : GATE   := AND_GATE;
                       -- AND means all predecessors must
                       --    be satisfied.
                       -- OR means one or more predecessors
                       --    must be satisfied.
                       --  Does not apply to leaf nodes.
    PRED      : PRED_NODE_PTR;
    SUCC      : SUCC_NODE_PTR;
    FORM      : NODE_FORM;
    MESSAGE   : DYNAMIC_STRING.UCSD_STRINGS;
  end record;

  type KNOWLEDGE_BASE is array (NODE_NUM range <>) of NODE;
                         -- ARRAY OF RECORDS
  type FLAGGED_NODES is array (INTEGER range <>)
             of NODE_NUM;                   -- Init. state
  function SIZE return INTEGER; -- ALLOWS SIZE OF GRAPH TO
                                -- BE READ AT RUN TIME.
end GRAPHS;
```

Figure 4. Graphs Package

N89-16351

AN APPROACH TO KNOWLEDGE STRUCTURING FOR ADVANCED PHASES OF THE TECHNICAL AND MANAGEMENT INFORMATION SYSTEM (TMIS)

H. T. Goranson, Sirius Incorporated
P.O. Box 9258, Virginia Beach, VA 23452

ABSTRACT: TMIS must employ an enlightened approach to its "object" structure, but basic issues in conceptual structuring remain to be resolved. Sirius outlines the necessary agenda and reports on progress toward solutions.

INTRODUCTION

Sirius is a small group which has traditionally focused on advanced solutions to troublesome issues in knowledge representation. Its prior work has primarily been for larger firms addressing problem spaces in the defense community. Recently, the group has been working in areas directly applicable to the specific problems of object/meta-data/knowledge representation faced by Space Station planners. A relatively mature paradigm is emerging which can be of use in the planning now being done.

THE SPACE STATION INFORMATION SYSTEM (SSIS)

SSIS has embraced the ADA philosophy, which is good. Study groups, particularly the one here at Clear Lake, have developed an agenda and strawman models. Insofar as we at Sirius have seen, the thinking is clear and has led to several early conclusions which we hope will not be diluted as activity increases and more interests become involved. In particular, there is a consensus that a Network Information System (NIS) entity be created and distributed in various incarnations throughout the system to work with (and allow a high level of independence of) "application" systems and communication systems.

Considerations of this NIS are intimately bound with another healthy emerging consensus: before architectures are considered, before coding, module or task planning is begun, even before the development agenda is formalized, a comprehension of the structure of the *representation universe* must be set. This universe must deliberately and circumspectly be engineered as a logical first step in the comprehensive life-cycle planning of the entire system.

This has truly never been faced squarely before, because prior requirements were narrow in comparison and allowed compromise. But SSIS-related issues are too broad, too deep, indeed too expensive *and important* to allow less than our best efforts in this matter.

The problem is: we immediately run up against the same fundamental, unresolved philosophical issues which have plagued the Knowledge Representation community for decades. There is traditional deadlocked contention among workers in the fields of Artificial Intelligence (AI), Programming Language (PL), and Database (DB) researchers which each represent justifiable perspectives.

The good news is that recent successful studies into the mathematical foundations of allied fields give us a whole new world of tools to work with, tools which allow us to transcend PL/DB/AI localizations and start to put our arms around the science required. This is fully in harmony with the ADA culture, and in fact our group has some ADA-based research underway in this area.

The bad news, at least so far, involves the Technical and Management Information Systems (TMIS), a system planned to support many facets of Space Station development.

TECHNICAL AND MANAGEMENT INFORMATION SYSTEM

While there are major differences between TMIS and SSIS, the TMIS object specification would rightly be placed *before* nearly all SSIS Engineering tasks (Engineering in the ADA sense). TMIS is now in a Phase I ad hoc configuration. Future phases will be competed shortly. What makes the TMIS problem so difficult, and so interesting is the manner in which it must evolve. Each user group will continue to work with his existing and planned environments, constantly adding to the data, information, and knowledge pools.

So the problem, the challenge of TMIS is to develop a conceptual universe which satisfies today's TMIS needs while building the object/representation/meta-data schema, which for simplicity we will call an Integrated Conceptual Environment (ICE). This ICE should be optimized for the emerging ADA centered SSIS, which includes AI interfaces, knowledge representation subsets, and anticipation of future (30 year horizon) technologies.

ISSUES: GENERAL

Basic issues involve ADA language and environment concerns, very large database theory and AI requirements.

Programming Language issues include a complete data and abstract data type specification, associative potential processing topology, as well as many inherent characteristics, such as fault recovery, fault tolerance and fault avoidance. With respect to issues in Programming Language (PL) design, a candidate representational formalism for SSIS characteristics should include constructs that allow a user to capture intuitions about the structure of the domain(s) of application - for example, intuitions about (or clustering operands for) the appropriate conceptualization of the objects, properties, and relations of the domain. The motivation for much of the prior research on incorporating abstract data typing facilities such as ANNA into the ADA programming language has resulted from this incidental requirement. Indeed, in the most general sense, it was this motivation that led to the development of high level programming languages, among them ADA, in the first place.

But neither the syntax nor the "general" semantics of first order languages will recognize the distinction between two types of abstracts, a frequently encountered phenomenon in an anticipated SSIS environment. Things of both types will simply be values of individual variables, and arguments of predicates (and function-symbols). To some extent, the analogue of this particular semantic inadequacy in programming languages, which is meant to be handled by the introduction of data types and typing facilities, *can* be treated within higher order calculus languages by introducing sortal quantification and by other tedious methods. But this is unacceptable for relating a potentially large abstract vocabulary in a KR context.

Our proposed approach extends a potential semantic capability of ADA into a semantic net operator set, in order to allow creation of a test bed for TMIS development, using current environments.

Concerning DB theory, one of the most striking features of the current explorations in ICEs is the attempt to deploy concepts and techniques from the field of Knowledge Representation, and especially from work on semantic networks, in combination with relational databases. However, one can see something of a paradox in the focus on current network-type formalisms. The backbones of most formalisms are taxonomic hierarchies, sometimes generalized to deal with multiple inheritance. A governing design requirement of TMIS is to capture semantic interdependencies (both inclusions and exclusions) among the various kinds of things (or "concepts") with which they have dealt. But the *explicit* embodiment of such interdependencies seems to be ruled out by the hierarchical relational model.

Given the structural limitations of existing ICE work, it is hard to see how such notions as type hierarchies, inheritance, and exclusion can be imported within the boundaries of the relational model. One proposed handling of hierarchies and inheritance is part of a reconstruction of database theory, from within the theory of first-order logic. TMIS runs the risk of adopting this default position without looking at a higher structural level.

It seems more natural to think about relational databases (especially as conceived of originally) as *models* of kinds of first and second order theories. It is instructional in the present context to propose extending the relational database concept in terms of locally finite dimensional cylindrical algebras. These would be algebras whose elements are sets of finite sequences of objects/attributes, rather than in terms of first or second order theories. We are in fact drawn more to the algebraic, morphological and semantic than to the syntactic, proof-theoretic, account of the relational model in this work. Of course, there are intimate connections between the morphological cylindrical calculus and algebras which our current research exploits, specifically within the unique TMIS universe.

ISSUES: NETS

There has been a long and continuing debate within the AI world on the relative merits and expressive power of representational languages based on Logic on one hand and semantic networks on the other. It is apparent that semantic nets have often been used as a notational rather than as a representational language. Logic, on the other hand, enjoys a well accepted semantics, but its support as a knowledge representation scheme is left unclear as to which aspects of which Logic it proposes to use.

In considering semantic networks, it should be noted that there has been considerable migration of techniques from semantic networks to data models in the VLD context. Net researchers are just now talking about multiple inheritance, whereas KB researchers have been involved for quite some time. One feature of semantic networks in a purely digital environment that needs to be considered by a data model is the genetics of cluster forces, over a KB life cycle.

A dispassionate observer may question why semantic networks haven't been used more directly for ICE purposes. Semantic networks and semantic data models are in a sense equally powerful, but they have been intended in the past for applications that have different and more linear characteristics. There is a trend in this research to deal with applications that involve an ever increasing number of types, and the gap in intended applications between data models and semantic networks seems to be narrowing. Especially interesting in this case are embryonic studies for photonic machines which appear to be providing a lead in this science of semantic nets.

TMIS will have to consider many possible uses for a database, only some of which would be of relevance to the ICE concerns of SSIS. Their approach is expected to be to determine the *least common denominator* of all these applications and then provide implementation. Although requirements for databases and VLDs have changed during the last few years, it is not obvious that the least common denominators for frameworks that handle photonic forms, visual and speech data, text messages, etc., are semantic networks.

The principals of Sirius, however, represent the philosophy of those who promote the use of nets without the restriction of logic-bound notations, and maintains the pre-eminence of this approach over the other, more constrained historical approaches.

KNOWLEDGE REPRESENTATION (KR) NETS AND LOGIC MATRICES

The theoretical limits of logic-based semantic nets are most clearly seen via properties of topologically isomorphic representations of the data structure. It is easily shown that the generalized map of each is a matrix (for the rule-based "truth table" approach) or a hierarchical structure (for the logic-driven "list processing approach"). The hierarchical geometries are often morphologically classed for queuing purposes and are often related, via pattern recognition layers, as the corners of an all logical space-filling regular three-dimensional tessellation. A commonly encountered example of this method of analysis can be seen in the oct-tree or quad-ternary-like types of machine-level conceptual structures now being investigated by Sirius and other workers in integrated photonics. By examining the morphology of existing Lisp-based methods, it can likewise be demonstrated that a ceiling of complexity results from commonplace simple hierarchical (read "list, stack, or string" processing) building techniques, as expressed in this conventional language syntax.

Consequently, some investigators have applied ingenious techniques to create, access, and manipulate structures of higher morphological complexity. Typical of the approaches is that of Lipski (1978) which advances the available geometry of the data lattices by placing matrix nodes in an ordering which emulates the hierarchical lattice of normal use. This, in fact, parallels a similar synthesis in generating grammars by Paz (1976). In turn, an associative transformational threading is allowed through the lattice, using recursive operators which give the combined structure great conceptual power.

But the theoretical limit on strictly hierarchical geometries remains low in morphological complexity and is limited to those of 3-D, orthagonal, regular tessellation.

Nets of much greater complexity have been used for decades by scientists and mathematicians, especially those involved in work for target image sensor fusion. Recently, the regular, infinite structures that are generatable by periodic recursion have been categorized by Goranson (1981), Burt/Wachman (1974), and Lalvani (1986). The approach to establishing a baseline vocabulary for the ICE comparative optimization studies selects a structure for the knowledge representation lattice based on the desired properties of the task and nature of the diversity of the source data and internal translative types.

This "Top Down" approach gives many advantages relevant to related *connection* biased theories. Major advantages for this proposed approach include related reduction of a meta-operator set (using primitive and recursive transforms) to a few mathematical symmetry operations to which we can apply several optimization techniques. Greatly increased density and organization of the information in the abstracted knowledge bases is achieved. This latter advantage allows a suitable level of abstraction to be combined with clustering of related elements (which can be seen as extended "frames"), a desirable effect which addresses suitability for local TMIS users. Memory requirements are greatly reduced in size. Elements which express cluster characteristics are typically quite large in number when narrow in scope and are well ordered for logical calculus interface. The resultant partitions are attractive for concurrent and parallel processing algorithms on a level low enough to include technologies projected for SSIS use.

IMPLICATIONS FOR THE PRESENT PROBLEM

The requirement is for an approach which:

a. Will provide a basis for synthetic studies of applicable technologies and methods across a wide spectrum, in short: a descriptive ICE nomenclature.

b. By virtue of providing a high level scientific description of the laws at work, will provide a unified means for integrating this diversity.

c. Allow an approach to optimizing ICE specification, design and hosting, provide for future generation computing requirements in terms of processing, large information sources, and reconfigurability.

d. Direct specification for hosting and testing of these issues.

GENERAL TOOLS

Sirius works with a set of morphological expressive laws which describe all possible global model schema. This morphology is currently embedded in a set of programs developed over time and currently used for internal research and is strictly within the ADA philosophy.

A specific tool cited, ALICE, described here as central to the approach is a method of machine hosting ICE morphological operators in ADA. This allows any high level semantic net to be quickly defined and tessellated among attributed space. While the geometries involved originate in Euclidean lattices, restrictions on three dimensional analogues disappear so the number of dimensions addressed is limited only by practical considerations of the size of the symmetry tensor used. Because TMIS requires operations that do not exclude direct microcoding in embedded systems, the number of dimensions is practically limited to 16 based on anticipated hardware, assuming that the lattice chosen has less than 3 symmetry characteristics. (Some work has been done by Sirius using linear infinite polyhedral geometries, hosted on fine-grained arrays, but the hardware required is specialized, and beyond the scope of this TMIS work.)

Major VLD and DB investigative interface tools are DAVID (from Goddard) and TIPS (from DoD). For determining the lattice geometries, the work of Lalvani and Burt forms the basis for long term studies by Sirius, and are employed here, following the symmetry operators proposed by Schoen (1970) and Wells (1977). The basic research of Brisson is the source for the multi-dimensional holomorphic linearizations required, as well as a source of understanding higher dimensional clustering effects when mapped to lower level matrices, which in turn follows a suggestion of Williams (1969).

Key to the approach is the intersymmetry minimal surface operator, an approach devised by Burt for the generic translation of intersymmetry groups. A complementary approach utilizing added dimensions of symmetry variables is used to annotate the entity attributes/relationship attribute from Lalvani.

The minimal surface operator methods themselves follow closely the trends established by Jenkins (1966), after Blatter (1971) and Voss. The actual trace algorithms were suggested and bounded by Rahimi (1972) and Barber (1970). The reflexive nature of that algorithm which reflects the Lalvani transpolyhedra loosely follows the technique applied by Yoshizawa (1982) to a similar problem in dynamics. Provisions for fractal doubt, not discussed here, come straight "out of the book" from McClure, as well as a few other specialists, not currently in the free world.

Lacking full-time access to supercomputer facilities, the present capability for VLD simulation and storage of multidimensional matrices that Sirius possesses is inadequate. A satisfactory, and expedient cost effective solution to this inadequacy has been found in the N-Dimensional Data Base System. This software exploits a special case of the symmetry storage case, namely the Generalized Balanced Ternary. It has been determined that the restrictions imposed on VLD research are far outweighed by the availability of this tool for the early phases of the effort. Rehosting overlays to TMIS host systems should be straight forward.

OVERVIEW OF ALICE

Sirius has been working this problem for some time. As described, a high-level conceptual language is required, for this type of effort reflecting the ponderous capabilities implicit in the problem. The result of Sirius' work is ALICE (ADA/Lattice Integrated Conceptual Environment). ALICE is coded in ADA, of course, and uses as operators, translation sets which are created by the abstraction tools described.

An outline of the features of ALICE:

a. The relational function set is extended into a primitive operator language. This language has a base logic of some order and has a mapping to the network lattice transforms.

b. These transforms have topological equivalences, a unique and interesting feature which allows the rules for operation among the fabric to be ambiguous when mapping "up", but precise when mapping "down", or out of abstracted spaces.

c. The topological equivalence also allows resolution into a few simple operations (which include photonic operators) which can be matrixed and solved by arithmetic array processors. This holds promise for high speed, concurrent AI processing using VHSIC; Reduced Environment Math processor arrays.

Having established a meta-language with a corresponding meta-net, and formal methodology the ICE researcher is faced with several exciting possibilities. Knowledge of any kind can be "fused" by simple procedures if the source calculus is descended from the universal structure employed.

For example, image data, graphic modeling information, engineering data (which may not necessarily be geometry dependent) and performance information can be all imposed on the <u>same</u> ICE. As relationships are established, the diverse information becomes fused, and is actually aggregated by virtual proximity; *Virtual Proximity* is defined as relation by a critical set of algebraic operators defined by the sliding threshold factors of the periodic lattice matrix.

BIBLIOGRAPHY

Lipski, W., "On Semantic Issues Connected with Incomplete Information Bases", TODS 4 (3), 1978.

Paz Azaria, "Multidimensional Parallel Rewriting Systems:, in "Automata, Languages, Development", ed Aristid Lindenmayer and Grzegorz Rosenberg, North Holland, 1976.

Goranson, H.T., "Periodic Recursion Dynamics in Large Integrated Conceptual Environments", Sirius, 1981

Wachman, A., M. Burt, and M. Kleinman, "Infinite Polyhedra", Technion, Haifa, Israel, 1974.

Lalvani, Haresh, "Structures on Hyperstructures", Privately Published, 1982.

Lalvani, Haresh, "Transpolyhedra", Privately Published, 1977.

Schoen, Alan H., "Infinite Periodic Minimal Surfaces Without Self Intersections", NASA TN D-5541, 1970.

Schoen, A.H., "Infinite Quasi-Regular Warped Polyhedra and Skewness of Regular Polygons", Not. Amer. Math. Soc., Vol. 15, 1968, pp 801.

Wells, A.F., "Three-Dimensional Nets and Polyhedra", (Chapters 19 & 20), Wiley, Interscience, 1977.

Brisson, David W., "A Definition of Distance and Method of Making Space-Time Measurements", Speculations in Science and Technology, Vol, 3, #5, 1980 pp 579-586.

Williams, Robert Edward, "Dimension as Level" in "Hierarchical Structures", ed. Whyte, Wilson and Wilson, American Elservier, 1969.

Jenkins, Howard and James Serrin, "Variational Problems of Minimal Surface Type III, the Dirichlet Problem with Infinite Data" Archive for Rational Mechanics and Analysis, Vol 21, #4, 1966.

Blatter, J., and E.W. Cheney, "Minimal Projections of Hyperplanes in Sequence Spaces" AFOSR-2130-71, Report NTIS AD 735 759 1971.

Rahimi, A. and R.W. Brokett, "Homotopic Classification of Minimal Periodic Realizations of Stationary Weighting Problems", SIAM Jour Applied Math 22, #3, pp 481-486, 1972.

Barber, M.N. and B.W. Ninham, "Random and Restricted Walks: Theory and Applications", Gordon and Breach, 1970.

Yoshizawa, T., "Stability Theory and the Existence of Periodic Solutions and Almost Periodic Solutions", Applied Mathematical Sciences #14, Springer-Verlag.

McClure, Donald E., "Image Models in Pattern Theory", Computer Graphics and Image Processing, Vol. 12, pp 309-325, 1980.

N89-16352

Ada* and Knowledge-based Systems: A Prototype Combining the Best of Both Worlds

ABSTRACT

In this paper, we describe a software architecture which facilitates the construction of distributed expert systems using Ada and selected knowledge-based systems. This architecture was utilized in the development of a Knowledge-based Maintenance Expert System (KNOMES) prototype for the Space Station Mobile Service Center (MSC). The KNOMES prototype monitors a simulated data stream from MSC sensors and built-in test equipment. It detects anomalies in the data and performs diagnosis to determine the cause.

The software architecture which supports the KNOMES prototype allows for the monitoring and diagnosis tasks to be performed concurrently. We have named the basic concept of this software architecture ACTORS, for Ada Cognitive Task ORganization Scheme. An individual ACTOR is a modular software unit which contains both standard data processing and artificial intelligence components.

A generic ACTOR module contains Ada packages for communicating with other ACTORs and accessing various data sources. It also includes an Ada package which acts as an interface between Ada and the knowledge-based component. For the first prototype, the knowledge-based portion was written in OPS5. By changing only the interface package, other ACTORs may be created with knowledge-based components written in LISP, PROLOG, or Ada (as knowledge-based systems are written for the language).

The knowledge-based component of an ACTOR determines the role it will play in a system. In our prototype we have an ACTOR to monitor the MSC data stream. The monitor can invoke other ACTORs as needed whose roles are to diagnose specific parts of the MSC. All of this activity is coordinated by a main ACTOR whose role is to oversee the interaction of the monitor and the diagnosis ACTORs.

*Ada is a trademark of the U. S. Government, (Ada Joint Program office)

David C. Brauer
McDonnell-Douglas Astronautics Company
Huntington Beach, California

N89-16353

Using Ada* to Implement the Operations Management System as a Community of Experts

ABSTRACT

This paper describes an architecture for the Space Station Operations Management System (OMS), consisting of a distributed expert system framework implemented in Ada. The motivation for such a scheme is based on the desire to integrate the very diverse elements of the OMS while taking maximum advantage of knowledge-based systems technology. This technology is needed not only to solve problems that are specific to various elements of the OMS, but also to help solve the problems of integration and evolutionary growth.

Part of the foundation work for an Ada-based distributed expert system was accomplished in the form of a proof-of-concept prototype for the KNOMES project (Knowledge-based Maintenance Expert System). This prototype successfully used concurrently active experts to accomplish monitoring and diagnosis for the Remote Manipulator System. We have named the basic concept of this software architecture ACTORS, for Ada Cognitive Task Organization Scheme. An individual ACTOR is a modular software unit which contains both standard data processing and artificial intelligence components.

The work accomplished in the KNOMES project and in similar efforts throughout the industry suggest that the maintenance task for Space Station can best be accomplished via a knowledge-based system approach. In addition, tasks such as planning and scheduling, as well as logistics management and payload activity management, may also benefit greatly from the use of knowledge-based reasoning in some form. Since all of the above applications form a subset of the complete OMS, it will therefore be necessary to include the potential for utilizing knowledge-based systems within the OMS -- if not at IOC, then during the growth of the station. In addition, it will be necessary for all of these OMS components to talk to each other.

It is when one considers the overall problem of integrating all of the OMS elements into a cooperative system that the AI solution stands out. By utilizing a distributed knowledge-based system as the framework for OMS, it will be possible to integrate those components which need to share information in an intelligent manner. This will be particularly crucial where processes need to be controlled in a hierarchical manner. One example is where a maintenance system for a particular subsystem needs to inform the scheduler about new constraints on the activities which that subsystem can perform.

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

Finally, there remains the question of implementing such a system within the confines of Ada. The proof that this can be done rests on the KNOMES prototype's use of ACTORS, on past implementations of AI languages using more traditional languages (eg., OPS-5 using BLISS), and on the implementation of a major commercial knowledge-based system environment in C. In fact, without the benefits of object-oriented programming as implemented in Ada, or the integration afforded by Ada's development constructs, the task for implementing a distributed knowledge-based system for OMS would be considerably more difficult.

M. S. Frank
McDonnell-Douglas Astronautics Company
Huntington Beach, California

N89-16354

APPLYING ADA TO BEECH STARSHIP AVIONICS

David W. Funk
Rockwell International
Cedar Rapids, Iowa

Abstract

Rockwell International's Collins Avionics Group has been active in the Ada* language development since 1978 when we participated in the design evaluation. As the language design solidified, it became evident that Ada offered advantages for avionics systems because of its support for modern software engineering principles and real-time applications. Starting in 1983, Collins developed an Ada programming support environment for two major avionics subsystems in the Beech Starship. The two subsystems include electronic flight instrument displays and the flight management computer system. Both these systems use multiple Intel 80186 microprocessors. The flight management computer provides flight planning, navigation displays, primary flight display of attitude as well as engine instruments and multi-function displays of checklists and other pilot advisory information. Together these systems represent nearly 80,000 lines of Ada source code and to date approximately 30 man years of effort. The Beech Starship avionic systems are in flight test now with expected FAA certification by the end of 1986.

Background

The Beech Starship is an entirely new turboprop airplane that will combine high performance and excellent fuel economy (see Table 1). The Starship with its composite construction, unconventional design and advanced avionics architecture presented a unique opportunity for Collins Avionics to pioneer the use of Ada in an airborne application.

Applying Ada to major subsystems of the Starship avionics offered several software engineering challenges. Except for the use of proven software design and testing methods all the other elements of software development were new. We started with new system/software requirements, a new development team, a new host computer environment (VAX), a new target computer environment (Intel 80186) and of course a new HOL Ada. It would be misleading to think the selection of Ada was the cause of all this newness because it was not. Given the new system requirements, all the other elements would be new, independent of the language.

* Ada is a registered trademark of the Department of Defense (AJPO).

TABLE 1

Starship Operating Characteristics

Max. Takeoff Weight	12,500 lbs.
Max. Altitude	41,000 feet
Cruise Speed	400 mph
Max. Occupants	10
Single Pilot IFR	

Starship APSE

The Ada Programming Support Environment (APSE) that Collins established for the Starship applications include a compiler, assembler/linker/loader, symbolic debugger, configuration manager, text editor and command language interpreter. The APSE is hosted on VAX (VMS) computers targeted to the Intel 80186 (see Figure 1). The components of the APSE are discussed below:

Compiler

Developed by Irving Compiler Corporation (formerly the Irvine Computer Science Corporation), the ICC compiler front end accepts an Ada source program and performs all lexical, syntactic, and semantic analysis. Under license to ICC, Collins developed the code generator for the Intel 8086 family of microprocessors. While the compiler is not validated by the Ada Joint Program Office, it only accepts valid Ada statements. At the beginning of the Starship projects it was determined that the ICC compiler was more than adequate to support the design constructs used in avionics software. This conclusion was reached by careful comparison with other HOL compilers in use at Collins. The compiler produces an assembly source file at the rate of 800 to 1000 Ada source lines per minute on a VAX 785. When the option is selected the compiler also produces a symbol table file for use by the symbolic debugger.

Configuration Manager

When the code is developed there must then be a method of keeping track of its evolution. This is partially the task of a configuration management tool. Source Tools by Oregon Software is used to manage a project's files by storing them in a library, tracking changes, and monitoring access to the library that contains the files. The Make function controls the efficient building of a software system by determining which components in the system have changed and then updating, or creating new versions of, only those files that depend on the changed components.

Symbolic Debugger

The testing phase is especially difficult for embedded computer applications. Because of this, Collins Avionics has developed a symbolic debugger which allows a developer to test a program at the Ada source code level on the target computer rather than at a lower machine code level. The debugger uses a database that is generated in part by the compiler, assembler and linker. This database is separate from the user program. This means that the user program need not be altered (it could even be in ROM) in order to use the debugger for testing.

The symbolic debugger which is also written in Ada is hosted on an IBM personal computer. The personal computer is fitted with cards that connect it via a cable to the target computer. These circuit cards provide additional RAM memory as well as control functions including signals to reset, halt, run and step the target computer, as well as facilities to examine and modify target memory location, an execution history buffer and address matching logic for breakpoints. The user interface to the debugger via the personal computer include commands for file manipulation, execution control, breakpoint, data manipulation as well as show and help commands.

Editor

The text editor currently being used is the EDT editor developed by DEC. It allows editing to be done in either full screen or line mode.

Interpreter

This task is currently performed by DEC's Digital Command Language (DCL) and its associated command processor. This provides a user with interactive program development, device and data file manipulation, and interactive and batch program execution and control.

Assembler/Linker/Loader

For the 80186, a VAX hosted cross assembler/linker package was purchased from Microtec Research. Collins developed programs were added to this package to provide data for the symbolic debugger.

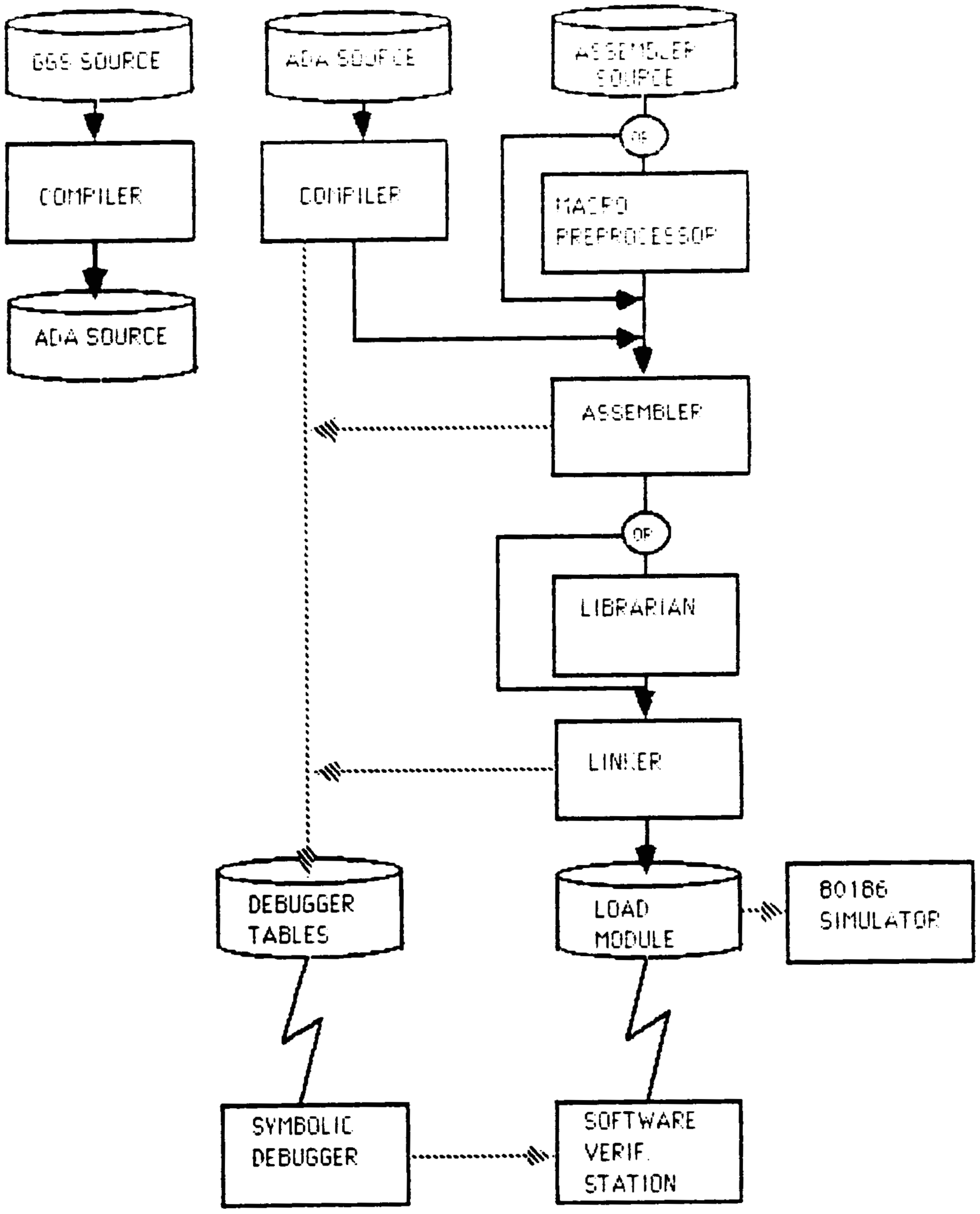
GGS Compiler

In order to help automate the generation of electronic flight display page formats, a general graphics system (GGS) compiler was added to the Starship APSE. As shown in Figure 1, the compiler accepts GGS source code and translates it into Ada source code. The GGS source is expressed in a language that allows description of a graphics object such as scales, pointers, numbers and letters as well as raster fill areas and stroke written areas. The GGS compiler which is written in Ada also runs on the VAX host computer.

ORIGINAL PAGE IS
OF POOR QUALITY

Ada Program Development Flow

FIGURE 1



Ada Applications

RTE

In order to support the embedded applications, Collins developed a Real-Time Executive (RTE) which is written in Ada. While the RTE is compatible with Ada, it uses a slightly different tasking model than the one directly supported by the language. A different model was chosen for two reasons. 1) It is a tasking model that Collins has used in many other embedded systems. 2) The ICC compiler did not completely support all of the Ada tasking features.

The RTE provides an interface from the application programs to the 80186 processor. Included in the processor resources are the CPU execution resource, interrupts and the timers. From the viewpoint of the application program, the following functions are supported: tasking, based on the concept of independent tasks, which share the CPU resources; time-based execution of tasks; event-based synchronization between tasks; external interrupt based execution of tasks and controlled access to resources such that independent tasks do not interfere with each other accessing shared resources.

A task is identified by stack, outer scope procedure, priority, and a four character name. Tasks are prioritized and may be activated by cyclic timer, event based signal or an external hardware interrupt. The RTE can support cyclic execution of tasks up to 1000 hertz. An Ada package may contain the outer procedure and stack for zero, one or more tasks, and procedures in a package may be executed by many different tasks. Table 2 lists the executive service routines that an application uses to interface with the RTE. The size of the RTE target code is approximately 10,000 bytes.

EFD

The Electronic Flight Displays (EFD) developed for Beech use 6 x 7 inch color CRT and completely integrated display processing. One display unit type is used in four applications in the cockpit. Table 3 indicates the applications and their functions. Each display unit is programmed with two applications to allow better redundancy and reversionary modes. In a two pilot cockpit, six display units, (four PFD/ND units and two EICAS/MFD units) are used . The Ada based applications execute on an pair of 80186 microprocessors. The first 80186 is used for application specific functions (such as PFD or ND) as well as input/output functions. The second processor is used for display control functions that are common to all display applications. Each processor uses a copy of the RTE.

FMS

The Flight Management System (FMS) provides a very flexible automatic multi-sensor navigation system which greatly reduces the pilot's workload. This system consists of a control display unit, a data base unit with a 3 1/2 inch floppy disk drive, and the flight management computer with a pair of 80186 processors. The Ada based applications implement the functions listed in Table 4. The software is partitioned on the two microprocessors in the following way. The first microprocessor provides control of the pages displayed on the CDU as well as control of the data base unit. The second microprocessor coupled with a floating point co-processor provides all of the navigation and performance computations. Some statistics about the EFD and FMS projects are summarized in Table 5.

TABLE 2

Executive Service Routines

Procedures:		
	START TASK	defines a task procedure, stack, priority, and name
	CHANGE PRIORITY	raise or lower a task priority
	ABORT TASK	stop task execution
	SET TIMER INTERRUPT RATE	defines cyclic task execution rate in hertz.
	WAIT FOR TIMER INTERRUPT	stalls task until next cyclic timer interrupt
	WAIT FOR (EVENT)	stalls task until a signaled event or external interrupt
	SIGNAL (EVENT)	used to synchronize with another task
	RESERVE (RESOURCE)	used to dedicate a shared resource to the calling task
	RELEASE (RESOURCE)	frees a shared resource (opposite of RESERVE)
Functions:		
	FULL SECONDS	returns current value of real time clock in seconds
	CENTI SECONDS	returns current value of real time clock in 0.01 seconds

TABLE 3

EFD Applications and Functions

PFD (Primary Flight Display)

Attitude
Flight Director
Lateral & Glideslope Deviation
Airspeed error
Alerts: marker beacon, decision height, altitude, ILS
Flight guidance modes (lateral & vertical)
Fault & off-normal annunciations
Reversionary "composite" PFD & ND

EICAS (Engine Instrument Crew Advisory System)
Torque
Prop RPM
Prop Sync
N_1
Fuel Flow
Oil temp, pressure
80 caution & advisory msg

ND (Navigation Display)

Heading
Selected heading/course/track
Lateral/vertical deviation
Bearings (ADF, VOR, WPT)
Distance/time to WPT
Groundspeed, windspeed
Flight plan with Navaids
Weather radar
Reversionary "composite" PFD & ND format

MFD (Multi-function Display)

Reversionary EICAS
Weather radar
Moving map (hdg up)
Planning map (north up)
Checklist (emergency & routine)
Nav status pages (pos, perf)
Diagnostic & maintenance data

TABLE 4

FMS Functions

(1) Statistical estimation of present position employing Kalman filtering techniques, utilizing all available sensor data.

(2) Automatic station selection, tuning, and management of a position fixing submode hierarchy, with provisions for pilot to intervene where appropriate.

(3) Adaptive leg-to-leg and off-course captures with g-limited steering law.

(4) Worldwide data base of VHF navigation aids, airport reference points, and published waypoints in numerous categories.

(5) Pilot creation of a large number of stored routes separate from the active flight plan, with provisions for off-aircraft creation and editing of the stored routes and ability to make trip planning calculations in flight.

(6) Calculations predictive of fuel remaining at destination

(7) Vertical Navigation function with deviation and steering outputs relative to fixed paths in space as may be defined in several ways.

TABLE 5

Ada Project Statistics

PROJECT	SOFTWARE ENGINEERS	LINES OF SOURCE CODE	TARGET CODE SIZE (BYTES)	HOST DISK (MEGABYTES)
EFD	8	31,000	318,000	164
FMS	12	51,000	470,000	189

Lessons Learned

Two and a half years of using Ada in real-time embedded systems have taught us some lessons about the application of the language. We chose a "walk first approach" in using Ada. Instead of trying to embrace the entire language with all of its new features, we chose to use a subset of the language that was similar to features of other high order languages we have used. This approach appeared to be wise in terms of training and in terms of bounding the number of variables in building a new system.

Training was accomplished by a combination of classroom work, textbook study and running examples on the VAX hosted APSE. The twelve hours of classroom training which was prepared and delivered by an in-house Ada expert, concentrated on the Ada concepts and features to be used by the Starship project. To complement the classroom work, all students were given a copy of the book "Software Engineering with Ada" by Grady Booch. Despite the fact that nearly every element was new in the software development process for these projects, we found software engineering productivity to average 200 delivered Ada source lines per man month. This is approximately equivalent to what we have experienced on other projects using more established programming support environments.

We discovered that the demands on the host computer system were greater than we had expected. This is in terms of both processing time and storage space. While the compiler is quite fast at 1000 lpm, adding the steps for the assembler, linker and debugger table generation reduce the average to 250 lpm. Add to this the fact that the configuration manager enforces recompilation of dependent packages and the result is longer processing times on the VAX host than we originally planned. Table 5 indicates the amount of host disk space required to support each project which is about twice what we have experienced on other HOL based avionics projects. The increase is explained by the additional files for configuration manager revision history, for the debugger database and the symbol tables needed by the compiler for Ada package specs and bodies. As far as code density on the target, we found the average of nine to ten bytes per Ada statement to be the same as other HOLs for the 80186.

We feel that Ada has brought some real benefits to the subsystems. Ada has provided the discipline and checks to allow program builds to work the first time in laboratory equipment. This greatly reduced debug time on the target computer. Ada has helped offer us more portable code between host system VAX's, personal computers and our target computers. And with the aid of symbolic debugging tools, our verification tasks are simplified.

N89-16355

Simulation of the Space Station Information System in Ada*

James R. Spiegel
Ford Aerospace & Communications Corporation
College Park, Maryland

INTRODUCTION

The Flexible Ada Simulation Tool (FAST) is a discrete event simulation language which is written in Ada. FAST has been used to simulate a number of options for ground data distribution of Space Station payload data. The results of these analyses include on-board buffer requirements due to the TDRSS zone of exclusion, as well as bandwidth versus buffer and bandwidth versus delay tradeoffs within the ground system.

The fact that the Ada language is used for implementation has allowed a number of useful interactive features to be built into FAST and has facilitated quick enhancement of its capabilities to support new modeling requirements. The use of tasks and packages has enabled the development of an interactive environment which allows the user to monitor and control the simulation. As a simulation is executing, a concurrent display task is updating pre-defined pages which contain simulation output statistics. A user command interface allows the user to pick from a number of display pages. This command interface also allows the user to interactively modify network parameters (e.g. number of servers or link bandwidth).

This paper discusses general simulation concepts, and then how these concepts were implemented in FAST. The FAST design is discussed, and it is pointed out how the use of the ADA language enabled the development of some significant advantages over classical FORTRAN based simulation languages. The advantages discussed are in the areas of efficiency, ease of debugging, and ease of integrating user code. The specific Ada language features which enable these advanced are discussed.

SIMULATION CONCEPTS

FAST is a general purpose discrete event simulation tool. Currently, there are a number of simulation languages that are recognized in the field of discrete event simulation. The list includes SLAM, GPSS, SIMSCRIPT, and others. The key feature that defines a "discrete event" simulation is that the state of the modelled system changes at discrete points in time. The simulation "language" automatically performs the task of keeping records of what events are planned to occur, and when they will

*Ada is a registered trademark of the U.S. Government (Ada Joint Programming Office) occur. The language also performs the task of maintaining statistics that describe the performance of the network elements. The job of the user of a simulation language is to model a given system within the constraints of the particular language being used.

The first step in the process is for the user make a abstraction of the system. Essentially, this means applying the terms of the simulation language such as "resource", "queue", and "traffic" to the user's particular problem. Figure 1 provides a table of different types of systems that may be modelled, and the associated meanings of each of the model elements.

One type of "network" which FAST has been used to model is the SSIS. In this case, the "traffic" entities are data packets, and the "servers" or "resources" are the communications links. Simulations were performed to answer such questions as:

How much bandwidth is needed ?

How long will data be delayed ?

What percentage of the time is the link busy?

For this example, the answer to the second and third questions are dependent on the first. The average wait time per packet is dependent on the link bandwith. The bandwidth is thus a "network parameter", while the delay times (queue statistics) and the link ("resource") utilization describe the system performance. The objective of a simulation activity is to predict the system performance as a function of the network parameters. This is usually done by performing a number of simulation "runs", while varying the network parameters. The result of each run is usually viewed as a point on a curve, and this curve describes the system performance as a function of input parameters.

The methodology used to implement an event-driven simulation is based on the concept of a future events queue. An event may be defined as any action or condition that changes the state of the system. Examples are when a data packet is generated, or when a transmission has been completed. The future events queue keeps a record of all of the events that may be planned. For example, when the transmission of a packet is initiated the time at which the transmission will be complete is calculated. This event is placed on the future events queue.

Each time an event occurs, a procedure is called that implements the logic associated with that event. This logic

SYSTEM	*TRAFFIC*	*RESOURCE*
SPACE STATION INFORMATION SYSTEM	DATA, COMMANDS	SPACE-GROUND LINKS GROUND-GROUND LINKS PROCESSORS
COMPUTER	DATA	PROCESSORS TASKS BUSSES DISKS
TELEPHONE	CALLS	CIRCUITS SWITCHES
MANUFACTURING	WIDGETS	WELDERS PAINTERS
BANK	CUSTOMER	TELLER
LOCAL AREA NETWORK	MESSAGES	BUSSES

Figure 1 - General Purpose Simulation Concepts

consists of decision making (is a link available?), updating the state of the system (the link is now busy), and performing calculations required to maintain statistics. When the processing for a given event is complete, then the future events queue is used to determine the next event.

One of the major problems in the area of simulation is efficiency. The process of discrete event simulation is inherently a Monte-Carlo process. This means that the input traffic is described by a statistical model. The simulation is thus performed using random inputs, and the statistics which describe the network performance are expected to converge in time. The number of events which need to be processed in order to achieve statistical convergence is both very large, and difficult to predict. The procedure usually adopted is to pick a safe duration, and to use this for all runs. This often results in two troublesome phenomena. The first is that more computer time is used that is actually necessary for a given run. The second is that the scope of the simulation activity is usually limited by the computing resource.

Another limitation of general purpose modelling languages is that they are usually not sufficient to model the complex interactions of a real world systems. Many simulation languages overcome this by allowing the user to write his own procedures. Mechanisms are provided for the user to write his own code (usually in FORTRAN), and integrate user written procedures in the model. The support available for this type of activity varies among languages, but in almost no cases can the support be considered "friendly". In most cases, the user is constrained to the use of a number of cryptic conventions in order to integrate his code. This process is both time-consuming and fraught with hazards. The bottom line is that one has to be a simulation "expert" in order to undertake such a task.

One final source of many headaches for users of simulation languages is the area of debugging. This includes both debugging of user written simulation routines (discussed in previous paragraph), and the debugging of models which do not work. Again, various languages provide various levels of support for this activity. As a minimum, most languages have the capability to list the names of what events occured and at what time. This results in a large listing which the user must search through in order to begin to understand where a problem is occuring. Once this information is found, it is sometimes useful, but oftentimes it does not shed enough light to solve the problem. When this happens the user is left little option other than staring long and hard at his input model, scratching his head, and trying to determine why he got the unexpected output. He may change one variable, rerun the model, and see what effect it had. When this fails to shed light, he will change others as deemed appropriate. This can be a very time consuming activity. Frustrated modellers have even been known to blame hardware.

FAST CONCEPTS

Three areas have just been described in which improvement is clearly welcome. These are :

- o EFFICIENCY
- o DEBUGGING
- o EASE OF INTEGRATING USER CODE

FAST has been designed with the objective of alleviating many of the obstacles which are encountered in these areas. In order to understand how these areas are addressed, it is necessary to first gain an appreciation for the overall FAST environment. This section provides a general description of the FAST environment, and then discusses the advances which have been recognized in these three areas.

FAST provides an unusually friendly environment in which to perform simulations. Figure 2 illustrates this "environment". FAST is designed to run interactively from a terminal. When FAST is running, most of the screen is dedicated to the display window. The user may specify which page is to be displayed by entering a "SET-PAGE" command in the input window. Figures 3 and 4 show the menus of user commands and pre-defined display pages which the user has to choose from. Figure 5 provides an example of one of these display pages. The other two windows are the error window, and the simulation state window. The error window is used when there is a syntax error in the user input, or when there is an error within the simulation run. The simulation state window displays whether a simulation is running, stepping, or suspended.

In a typical use of FAST, the user runs a simulation and monitors a statistic of interest. When the statistic has converged, the user changes the network parameters, and a new simulation "run" is started. This environment presents a number of advantages, the most important of which is that the user is able to observe the statistic as it is updated in accord with the progress of the system. The period of time required to obtained confidence in the results is significantly reduced.

```
+-----------------------------------------------------------------------------+
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                            DISPLAY WINDOW                                   |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
+-----------------------------------------------------------------------------+
| SIMULATION STATE WINDOW              |     ERROR WINDOW                     |
+--------------------------------------+--------------------------------------+
| INPUT WINDOW                         |
+--------------------------------------+
```

Figure 2 - The FAST Environment

```
+-----------------------------------------------------------------------------+
|State menu commands:                                                         |
|                                                                             |
|Help                                           -- Display this help screen   |
|Help  <Set_Page>                               -- Display page selection help|
|List                                           -- List all state files       |
|New  [<filename>]                              -- Start a new state file     |
|Open [<filename>]                              -- Open a state file          |
|Set_Speed <number of seconds from 1 to 60>     -- Set refresh rate of display |
|Set_Duration       <simulation time>           -- Set duration of simulation |
|Set_Queue_Size     <queue number>    <size>    -- Set size of one queue      |
|Set_Resource_Size <resource number> <size>     -- Set size of one resource   |
|Flush                                          -- Flushes Statistics         |
|SAVE                                           -- Saves the current state file|
|Close                                          -- Close a state file         |
|Print                                          -- Print Simulation Results   |
|Quit                                           -- Return to limit menu       |
+-----------------------------------------------------------------------------+
 Execution Suspended.
 Input :
```

Figure 3 - Input Command Menu

```
+-----------------------------------------------------------------------------+
|Page Selection State menu commands:                                          |
|                                                                             |
|Set_Page Queue_Resource_Summary          -- Display queue / resource stats   |
|Set_Page Mark_Summary                    -- Display mark statistics          |
|Set_Page Limit_Summary                   -- Display limit statistics         |
|Set_Page Queue_Resource <queue number>   -- Display statistics on one queue  |
|Set_Page Queue          <queue number>   -- Display a queue                  |
|Set_Page Future_Events_Queue [<number>]  -- Display Future Events Queue      |
|Set_Page Mark           <mark number>    -- Display statistics on one mark   |
|Set_Page Passport_Summary  [<number>]    -- Display status of all passports  |
|Set_Page Active Passports  [<number>]    -- Display status of active passports|
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
+-----------------------------------------------------------------------------+

 Input :
```

Figure 4 - Display Page Menu

```
+-----------------------------------------------------------------------------+
|Queue and resource summary               Current simulation time is 10000.0000|
|                                                                             |
|Queue  Arrivals  Avg Length  Avg Wait  Avg Resource Usage                    |
|-----  --------  ----------  --------  ------------------                    |
|    1      8153      1.0111    1.2401              0.4144                    |
|    2      8251      0.0529    0.0642              0.4282                    |
|    3      8153      4.3402    5.3238              2.1731                    |
|    4       102      1.3550  133.8373              1.0000                    |
|    5       600     27.1731  465.1406              1.0000                    |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
|                                                                             |
+-----------------------------------------------------------------------------+
 Execution Suspended
 Input :
```

Figure 5 - Sample Queue-Resource-Summary Page

The user may control the execution of the simulation through the use of "START", "STEP", "STOP", and "RESTART" commands. In addition, he may alter network parameters using the "SET-QUEUE-SIZE" or "SET-RESOURCE" command. The result of these ccapabilities is that an environment is provided in which the user may monitor and control the simulation process.

EFFICIENCY

The ability to provide the capability to build and monitor display pages was facilitated by the use of Ada tasking. The FAST system consists of a number of tasks. One of these is the simulation task, which performs the actual network simulation. In addition, there are tasks for displaying pages, as well as tasks to interact with the user.

There is no way of getting around the fact that Monte-Carlo simulation takes a long time in order to achieve statistical convergence. What FAST does provide, however, is a mechanism to monitor the statistics in question. The user may monitor a statistic during a simulation run, and when the statistic has stabilized to the user's satisfaction, the run may be stopped. This provides two advantages. The first is that a confidence range may be established. The second is that the user does not have to guess how long to run the simulation, thus saving a lot of personal and CPU time.

DEBUGGING TOOLS

Clearly, for an event driven simulation, the future events queue is vital to the inner workings of the simulation. The ability to view the future events queue on an event by event basis is extremely valuable for debugging purposes. One of the major advantages of FAST is that it does allow visibility into the "internal" structures of the simulation. These include both a "Future-Events-Queue" page, as well as an "Active-Passports" page. (A passport is a record that is used to keep track of the traffic entities as they flow through the system). The Future-Events-Queue lists which passports are scheduled to be activated, and when. The Active-Passports page describes where within the network each passport resides, as well as additional information about the passport. The combination of these two pages provides significant detail regarding the state of the system. FAST also includes a feature called step mode, which allows the user to instruct the model to process only one event at a time. Using the pages in conjunction with the step mode the user is able to observe the very fine details of the system, and can do so at whatever level is deemed appropriate for determining exactly how the simulation is progressing.

In addition to the features that have been described, the process of debugging is reinforced by an error management philosophy that takes advantage of Ada's exception handling. If, in the process of a simulation, a logical simulation error is encountered (such as a queue overflow), this error is managed as an exception. The simulation is suspended, and an error message is displayed in the error window that describes the error. At this point, the user is able to investigate why this error has occured. All of the simulation structures are still in tact, so the user may use the display capability to observe any of the pre-defined pages.

One proven debugging technique is to use the "SET-DURATION" command to a time just prior to the simulation error. A "RESTART" command will then cause the simulation to run to a point just before the error occurs. The user may now proceed using the STEP command to determine exactly when, where, and why the error occured. Clearly, such a capability is invaluable in the debugging process.

<u>EASE OF INTEGRATING USER CODE</u>

FAST has been designed in such a way that makes adding user modules safe, efficient, and easy. This is due to the fact that an object oriented design has been implemented which not only protects the system from the user, but also provides maximum support for the user.

As an example, there is a "QUEUE" package which contains the data structures which are used to model the queues, and all of the procedures and functions which operate on queues(e.g. REQUEST, RELEASE). Within the queue package, all of the logic which is needed to model the queue (First-In-First-Out) is coded. All additional effort which is required in order to implement these functions is provided by support packages. All of the queue length statistics (average, standard deviation, maximum, minimum) are maintained by a statistics package. Within the queue package, whenever the queue length is altered, a message is sent to the "STAT" package. Similarly, communications between instances of queue and passport structures is through a message oriented protocol similar to Smalltalk in nature. Real-time displays are implemented through messages to a window manager (also implemented in Ada).

All of these support packages which are currently used by the existing FAST packages are available for user written packages. This means that the inclusion of user packages is both safe and efficient. In addition, the debugging capabilities significantly reduce the time necessary to test and integrate large models. Finally, the user packages are written in Ada, and are thus blessed with the inherent advantages therein.

The use of object oriented design has already provided significant efficiencies in the development of the FAST system. In addition to all of the classical arguments espoused by the proponents of object oriented design, the methodology lends itself particularly well to the implementation of a simulation language. Specifically, objects that are built to model elements are limited in scope and complexity to the problem of modelling the logic of that element.

CONCLUSION

FAST uses the capbilities of the Ada language (packages, tasking, and exception handling) in order to enhance a classical simulation tool by providing an interactive, friendly simulation environment. The result is a tool which is easy for a beginner to use, and significantly increase the productivity of an experienced network simulation specialist.

DESIGNING WITH ADA* FOR SATELLITE SIMULATION: A CASE STUDY

W. W. Agresti, V. E. Church, D. N. Card, P. L. Lo
Computer Sciences Corporation**

ABSTRACT

A FORTRAN-oriented and an Ada-oriented design for the same system are compared to learn whether an essentially different design was produced using Ada. The designs were produced by an experiment that involves the parallel development of software for a spacecraft dynamics simulator. Design differences are identified in the use of abstractions, system structure, and simulator operations. Although the designs were significantly different, this result may be influenced by some special characteristics discussed in the paper.

INTRODUCTION

Some early experiences using Ada for scientific applications (e.g., [1]) showed that the design of the Ada system "looked like a FORTRAN design." As part of an experiment on the effectiveness of Ada, the experiment planners identified the following factors that were believed to be prerequisites for obtaining a new design, one that would take full advantage of Ada features:

- The opportunity to set aside previous designs for the system and work directly from system requirements
- Training in design methods that exploit Ada's capabilities
- The encouragement to explore these new design methods

The purpose of this paper is to address the following question:

When these prerequisites were satisfied, was a different design produced?

The experiment in progress is being conducted by the Software Engineering Laboratory (SEL) [2] of the National Aeronautics and Space Administration's Goddard Space Flight Center (NASA/GSFC). NASA/GSFC and Computer Sciences Corporation (CSC) are cosponsors of the experiment, which is supported by personnel from all

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

**Authors' Address: Computer Sciences Corporation, System Sciences Division, 8728 Colesville Road, Silver Spring, Maryland 20910 three SEL participating organizations (NASA/GSFC, CSC, and the University of Maryland).

The objective of the overall experiment is to determine the effectiveness of Ada for flight dynamics software development at NASA/GSFC. ([2] describes the characteristics of this environment.) The experiment, begun in January 1985, consists of the parallel development, in FORTRAN and Ada, of the attitude dynamics simulator for the Gamma Ray Observatory (GRO) spacecraft. When completed, the system is expected to comprise approximately 40,000 source lines of code to execute on a DEC VAX-11/780 computer. Additional information about the experiment is presented in [3].

Although the FORTRAN and Ada development teams are proceeding in parallel, the FORTRAN team is further along, due, in part, to the time necessary to train the Ada team in the Ada language and design methods. Both teams have completed the critical design review. This paper reports on a preliminary review of the design processes and products of both teams in order to address the question of interest. The design problem is discussed, an overview of the designs is presented, design processes and products are compared, and the results and their implication for answering the question are summarized.

THE DESIGN PROBLEM

The purpose of the GRO dynamics simulator is to test and evaluate GRO flight software under conditions that simulate the expected in-flight environment as closely as possible [4]. The simulator is represented as a control problem in Figure 1. The right side of the figure models the onboard computer (OBC) flight software. The OBC Model uses sensor data provided by the Truth Model to determine the estimated attitude. Comparing the estimated attitude to the desired spacecraft attitude, the OBC determines the attitude error. Control laws are modeled within the OBC to generate attitude actuator commands that will reduce the attitude error.

The Truth Model, the left side of Figure 1, simulates the response of the attitude hardware. The Truth Model updates and interpolates the spacecraft ephemeris and environmental torques, integrates the spacecraft equations of motion, and generates the true attitude of GRO. The Truth Model produces sensor data corresponding to the attitude, for use by the OBC Model.

Both teams have the task of designing and developing software to simulate the attitude dynamics and control shown in Figure 1. An additional requirement on the FORTRAN team is to extract its Truth Model and integrate it with the Goddard GRO Simulator (GGS), a real-time simulator of the GRO OBC flight software.

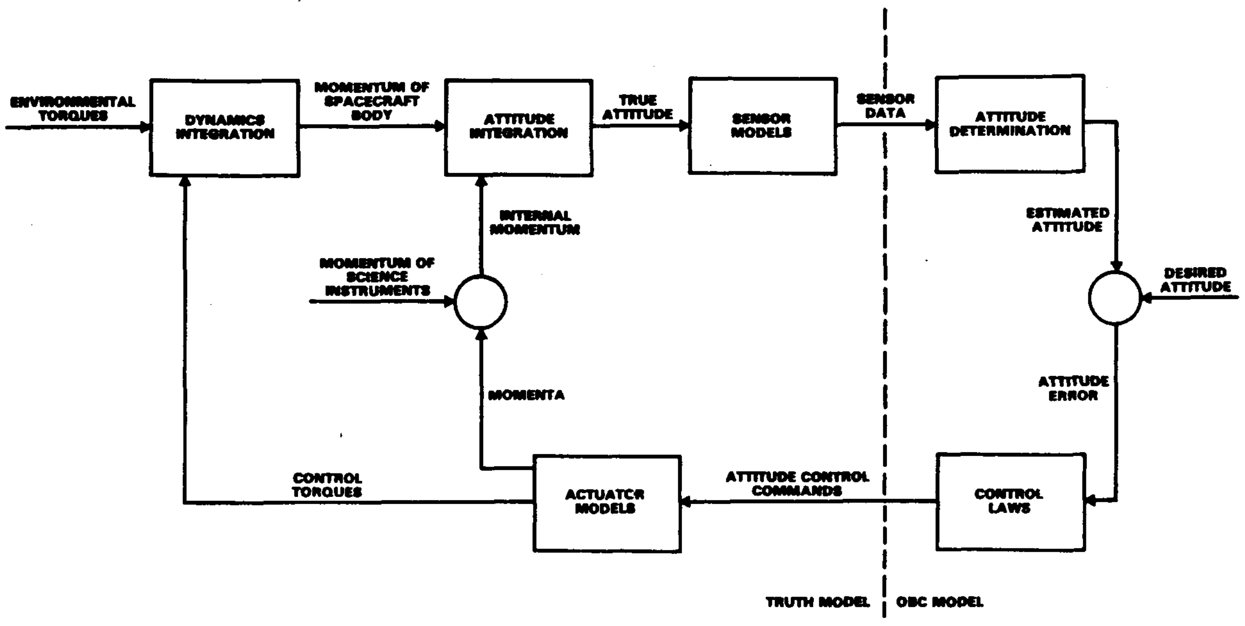


Figure 1. GRO Dynamics Simulator as a Control Problem

OVERVIEW OF THE DESIGNS

In this high-level look at each design, the overall system structure and the external and internal data flows are discussed. Some simple quantitative measures are extracted from each design.

System Structure

A top-level system diagram for each design is shown in Figures 2 and 3. To facilitate comparison, the identical system input and output objects are placed at the top and bottom, respectively, of each figure. The FORTRAN system consists of the five subsystems in the middle of Figure 2. The Ada system is the product of a design method (discussed below) that differs from the FORTRAN team method. So, although "subsystem" will be used to refer to the major Ada units, they are, in fact, Ada packages. Furthermore, the simulation support subsystem in Figure 3 is really a collection of three Ada packages for the simulation timer, parameters, and ground commands. The Ada system appears in Figure 3 as five subsystems only to invite comparison with FORTRAN regarding the high-level data flow.

The FORTRAN system is composed of three distinct programs: Profile, Postprocessor, and Simulator (Truth Model, OBC Model, and Simulation Control-I/O). As separate programs, each interacts with the user, as shown by the external data flows in Figure 2. The assignment of processing functions to each subsystem is shown in Figure 4 for both the FORTRAN and Ada systems.

ORIGINAL PAGE IS
OF POOR QUALITY

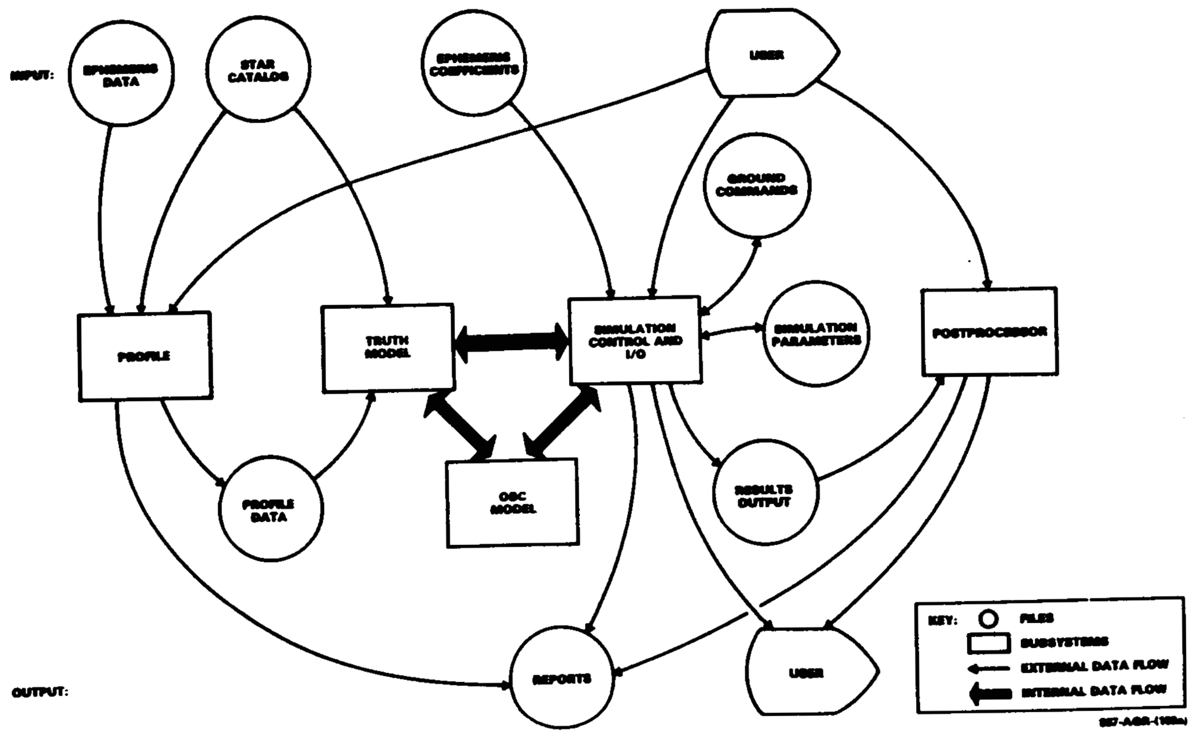


Figure 2. FORTRAN System Diagram

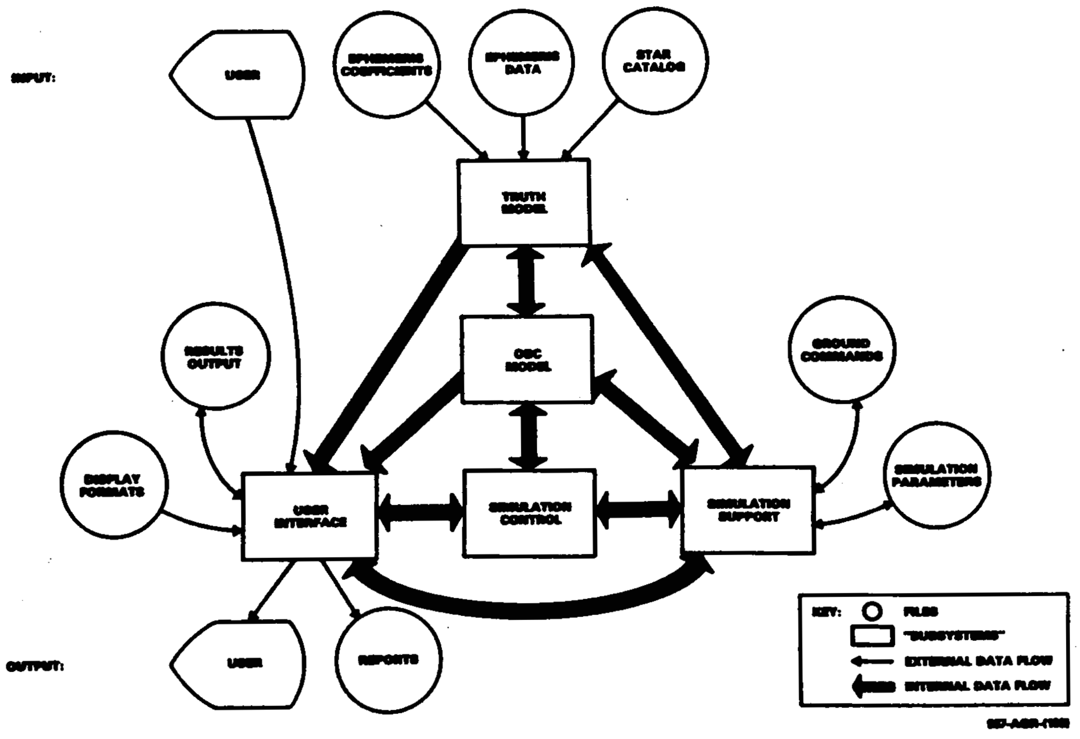


Figure 3. Ada System Diagram

ORIGINAL PAGE IS
OF POOR QUALITY

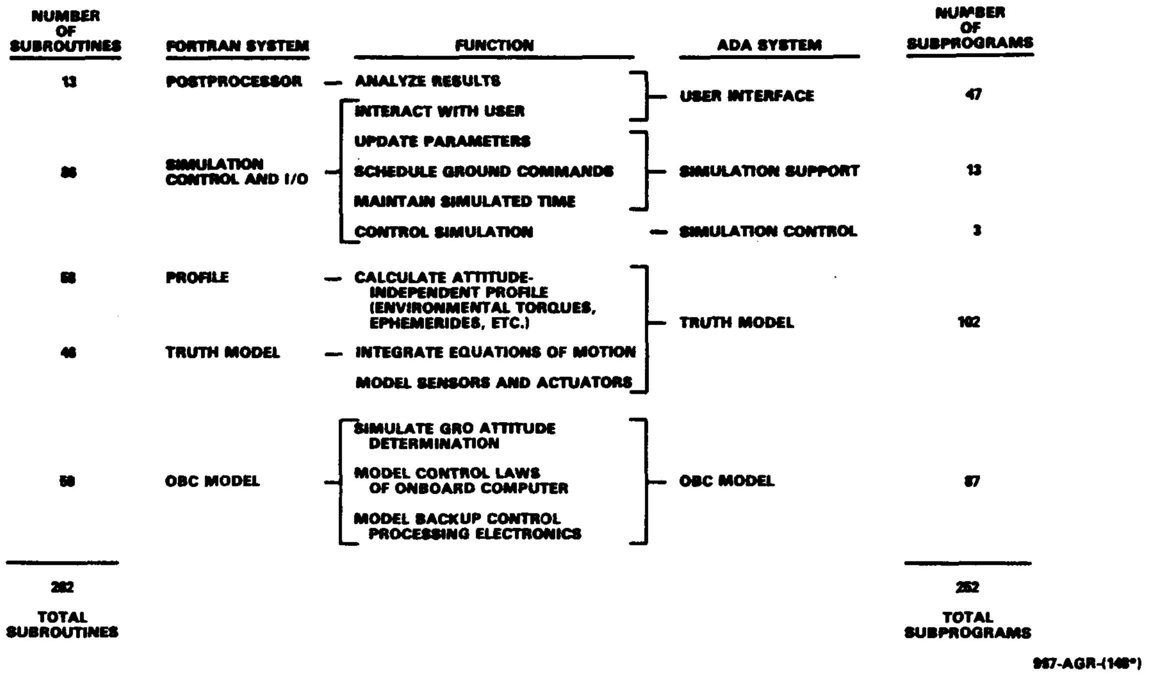


Figure 4. Allocation of Functions Among Subsystems

The Ada system is designed as a single program, with each subsystem performing the functions listed in Figure 4. The OBC Model is functionally similar to its FORTRAN counterpart. The Ada Truth Model incorporates the processing performed in the FORTRAN Profile in addition to the FORTRAN Truth Model. (The FORTRAN user has the option of choosing not to use Profile and having those calculations performed in the Truth Model, thereby mirroring the Ada design.) The Ada design pulls apart the simulation control functions from the User Interface; these processing elements are combined in the FORTRAN design. However, the User Interface in Ada includes the results processing that, in FORTRAN, is delegated to a separate program, the Postprocessor. Both designs have major units named Truth Model and OBC Model to reflect the underlying control problem illustrated in Figure 1.

External Data Flow

Both designs in Figures 2 and 3 show communication with nine external objects (files or devices). Eight of the nine are identical, the difference being the profile data file in FORTRAN and the display format file in Ada. The FORTRAN design requires the profile data file to decouple the Profile and Truth Model processing. The use of a display format file in the Ada design is motivated by reusability considerations. By keeping the detailed formats of menus and displays on an external file, the user interface is easier to reuse on a future simulator.

The number of external data flows is greater in the FORTRAN design, as shown in Table 1. Most of the additional data flows arise from the separation of the FORTRAN design into three programs, requiring more data flows to and from the user and distinct data flows to the profile data and results output files that decouple the programs. Also, as shown in Figure 2, the star catalog external file is required in both Profile and the Truth Model.

Table 1. Simple Quantitative Design Characteristics

CHARACTERISTIC	FORTRAN DESIGN	ADA DESIGN
SEPARATE PROGRAMS	3	1
TASKS	5 (IN SIMULATOR PROGRAM)	5
EXTERNAL ENTITIES	9	9
EXTERNAL DATA FLOWS	18	10
INTERNAL DATA FLOWS	3	9
SUBROUTINES/SUBPROGRAMS	262	252
PACKAGES	—	104

0217-(000)/86

The Ada design (Figure 3) involves the minimum number of external data flows. The details of accessing each file are confined to a single subsystem.

Internal Data Flow

Table 1 shows that the Ada design has nine internal data flows, versus three for the FORTRAN design. Of course, no more internal data flows are possible in the FORTRAN case because Profile and the Postprocessor are separate programs. The three remaining subsystems in the FORTRAN design exchange data with one another via COMMON blocks. (Although the use of COMMON has been criticized, empirical results from the flight dynamics environment has shown it to be effective [5].)

Although the number of distinct data flows (connections) between subsystems is greater in Ada, fewer data items pass over these connections than in FORTRAN. An example will show how various Ada language features help to reduce the proliferation of data item names.

Both designs provide for the recording of simulation analysis results. In FORTRAN (Figure 2), these results pass from the

Truth Model and OBC Model via COMMON to the Simulation Control-I/O Subsystem, which writes them to the external results output file. In Ada (Figure 3), the internal data flows from the Truth Model, OBC Model, and Simulation Control carry results data to the User Interface, which writes them to the results output file.

In the FORTRAN design, the results data record comprises 43 distinct variable names. In Ada, the results are passed under a single identifier, Results_Data, when a procedure, Put_Results_Data, in the User Interface is called by the Truth Model, OBC Model, or Simulation Control. This reduction in the number of identifiers is possible because of the use of Ada's variant record feature. In the example, Results_Data can be either an executed ground command, parameter update, error message, or analysis result. In Ada, the user can declare Results_Data as type RESULT, defined as a record type with a variant part as follows:

```
type RESULT_KIND is (Error_Msg, Log_Command, Results,
                     Parameters);

type RESULT (Kind:  RESULT_KIND:=Results) is
  record
    case Kind is
      when Error_Msg | Log_Command =>
        Result_Line:  STRING (1...80);
      when Results | Parameters =>
        Result_Rec:  PARAM_RESULT;
    end case;
  end record;
```

Because of such features, the count of data items is consistently lower over the Ada data flows than over the FORTRAN data flows.

COMPARING DESIGN PROCESSES

Differences in the design processes help to explain the differences in the delivered design products of the FORTRAN and Ada teams. Two aspects of the design process--critical design "drivers" and the use of design abstractions--will be examined.

Design Drivers

The design drivers--critical characteristics that strongly influence design decisions--are different for the two teams. The FORTRAN team was influenced by its real-time processing requirement, previous designs, and schedule concerns. The Ada team was influenced by its training in alternative design methods and the opportunity to apply those methods.

Although the basic requirements for each team are identical, the FORTRAN team has a real-time requirement, noted earlier, to integrate its Truth Model Subsystem with the Goddard GRO Simulator. To help ensure that the Truth Model will complete its processing in time to meet this requirement, the FORTRAN design removes those computations that are not strongly attitude dependent from the Truth Model to a separate Profile Program. Then, instead of performing these calculations (such as environmental torque and magnetic field) each iteration, the Truth Model can simply read the necessary values from the Profile data set (as shown in Figure 2). This separation of the Profile calculations from the Truth Model is further encouraged by the previous designs of dynamics simulators in FORTRAN, which also had separate Profile Programs. The FORTRAN design also provides the option, for greater accuracy, of performing the Profile calculations within the Truth Model.

The Ada design, not required to meet the real-time constraint in this experiment, includes in its Truth Model the calculations performed in the FORTRAN Profile Program and FORTRAN Truth Model. It will be of interest later to test whether the real-time requirement can be met by the Ada design and by the FORTRAN design under the option of performing Profile calculations inside its Truth Model.

A strong driver of the FORTRAN design is the presence of a previous design, used successfully on past simulators. The partitioning into subsystems in Figure 2 is identical to that of previous simulators. With this legacy, the interfaces between subsystems--a frequent problem area with original designs--are clarified early in the project. With the interfaces relatively clear, the subsystems can be assigned to individuals or small subgroups for detailed design and implementation with the "design envelope" fairly well established.

The Ada design was intended to be an independent one, free of the influence of past simulator designs. The subsystems that evolved were the product of lengthy design discussions. The similarity of the Ada subsystems to those in FORTRAN owes more to both designs reflecting the underlying control problem of Figure 1, rather than the Ada design copying the FORTRAN design.

The schedule constraints on the teams were different. To help explain this difference, consider that the dynamics simulator is a routine element of the set of ground support software for a satellite mission. The entire complement of software has rigid schedule constraints derived from launch dates. FORTRAN has been used in the past and is being used now for the GRO attitude ground support software. In such an environment, it is natural that the FORTRAN team was perceived as building the real, operational software, even though the Ada product is also expected to pass acceptance testing and to perform in an operational environment.

The FORTRAN team generally had more schedule pressure than did the Ada team, and this difference affected the design products and methods. Both teams were charged with developing operational software, but the Ada team was also encouraged to try Ada-related design methods as a way of understanding their usefulness in the flight dynamics environment. The FORTRAN team had more exclusively practical concerns of meeting the development schedule.

Design Abstractions

The use of abstraction was also different for each team. The FORTRAN design products provide evidence of the procedural abstraction carried forward from earlier designs. An individual subroutine may be thought of as a black box that will, for specified values of its input variables, produce the same specific output values every time it is invoked. The input and output quantities are transmitted via argument lists or COMMON. This procedural abstraction can also be used at higher levels in the system. For example, the Truth Model is a procedural abstraction possessing an identifiable function (computing the current attitude state of the spacecraft), specific input quantities (primarily parameter values and actuator commands), and specific output quantities (primarily sensor data reflecting the time attitude state).

The FORTRAN design also has elements of being object oriented. Functional processing at the lower levels is organized around objects in the problem domain such as specific sensors and actuators. For example, the Truth Model contains a sensor modeling component that calls seven routines: one for each sensor type. Anyone making a code modification due to a requirement change relating to the fine Sun sensor will find a subroutine, FSSMOD, described as modeling the fine Sun sensor. The use of COMMON also reflects an orientation to objects. For example, one COMMON block holds gyro parameters; another has FSS parameters; and so on.

Concurrent processes are used in the FORTRAN design to model the concurrency that exists in the operational use of the simulator. For example, an analyst may interrupt the processing to change the value of a parameter. System services of the DEC VAX-11/780 VMS operating system are used to implement the concurrent processes. Both the object-oriented features and the use of concurrency are characteristics of past FORTRAN simulators, demonstrating that reuse of design is the operative high-level approach in the FORTRAN design.

The Ada design process was significantly different from that of the FORTRAN team. The differences begin to emerge even before the design phase of the project.

The functional specifications and requirements document [4] for the GRO dynamics simulator is influenced by the design legacy of dynamics simulators developed within the organization. For example, the document is organized by major subsystem because that particular partitioning into subsystems (Figure 2) has persisted through several simulator project teams. In effect, the highest level design is completed during the requirements analysis phase.

This encroachment of design on requirements actually provides a welcome headstart to a team who will be following that design and taking maximum advantage of the existing code based on that design. While such a document fit in well with the projected work of the FORTRAN team, it was not as helpful to the Ada team, who wanted to produce an independent design, uninfluenced by previous simulator designs.

A way out of this dilemma--the influence of the previous design present in the requirements--was to recast the requirements in a different form. The Ada team developed a specification for the dynamics simulator using the Composite Specification Model (CSM) [6], which represents a system from the functional, dynamic, and contextual views. Recasting the system requirements using CSM served other purposes as well: It provided a testbed for the CSM as a specification tool, and it allowed the Ada team, who was relatively inexperienced in the application area, to analyze the system requirements in a systematic manner. The result of this exercise was a specification document [7] and a better understanding of the needs of the system. For example, included in [7] are PDL-like process specifications describing the required functional processing. The specification succeeded in removing the inherited design from the system requirements and served as a starting point for the Ada design.

The Ada language itself influenced the design team because the team members knew that useful design abstractions could be represented in Ada. The team had been exposed to object-oriented design, tne process abstraction methodology, and other approaches during their training program, which included the development in Ada of a 5700-line training exercise [3]. The principal design abstractions used by the team were the state machine abstraction and the representation of the system according to the orthogonal views of a seniority hierarchy and a parent-child hierarchy [8]. The state machines are conveniently implemented as Ada packages consisting of internal state data and a group of related procedures that operate on that state data. The Ada design product reflects this approach; the design includes 104 packages and 69 sets of state variables.

An instance of the seniority hierarchy is shown in Figure 5. The team's design approach is to build the system as layers of virtual machines [9]. For example, Figure 5 shows that the OBC package is <u>senior</u> to the Truth Model package. The arc between the two packages shows that OBC uses operations (subprograms) of the Truth Model. Arcs do not go from a package to one that is above it. In this way, each diagram expresses the relative seniority of the packages [10]. The orthogonal parent-child (or inclusion) hierarchy provides for a package (like one of those in Figure 5) to be represented on a separate diagram in terms of its constituent elements; for example, subprograms, other packages, and state data.

ORIGINAL PAGE IS
OF POOR QUALITY

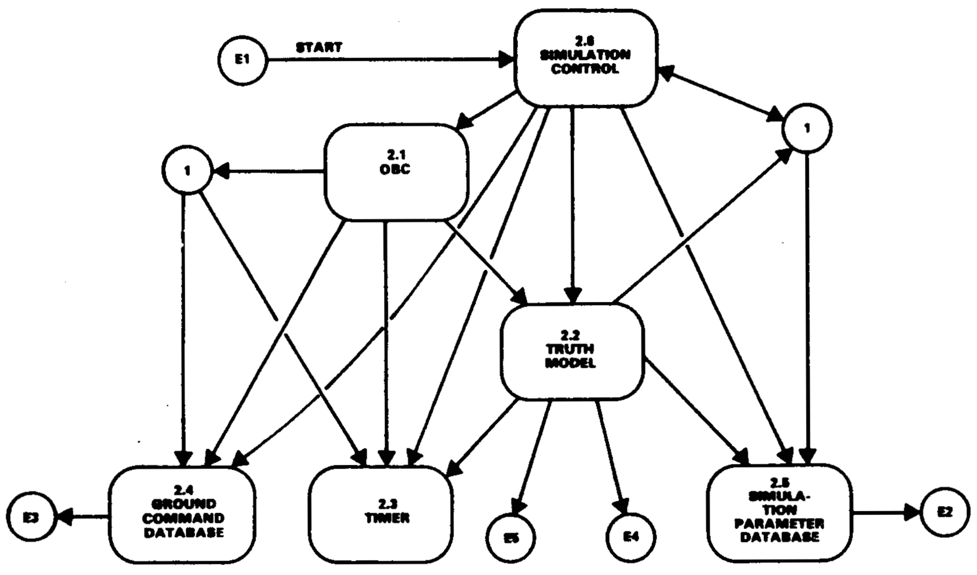


Figure 5. Ada Design: Seniority Hierarchy of Packages

In summary, the Ada team was able to use effective design abstractions because they were confident that these abstractions could be preserved in an Ada implementation.

COMPARING DESIGN PRODUCTS

The design documents were examined to determine any significant differences. Some differences were noted earlier: the FORTRAN design involving three programs; the different assignment of functional processing to subsystems; and the data flow. Review of the design documents revealed two more fundamental differences in the basic operation of each simulator, as specified by the designs. These key differences can be shown by tracing the operation of each simulator.

Figure 6 shows the logical relationships among the five tasks that constitute the FORTRAN simulator program (i.e., excluding Profile and the Postprocessor). The task called GROSS in Figure 6 is the main process started by the user via a RUN command. GROSS remains an active process throughout the simulation run, displaying a menu of user options at the user's terminal and remaining ready to respond to a user request.

The SIMCON process, created by GROSS, controls the simulation. As suggested by the control loop in Figure 1, the simulation involves iterating over the Truth Model and the OBC Model. SIMCON directs this iteration. SIMCON wakes up the Truth Model (TM) process, which computes the attitude state and deposits the corresponding sensor data into a global COMMON section. When TM is finished, it goes into hibernation, setting an event flag that signals SIMCON to wake up the OBC process. OBC obtains the current sensor data left by TM, models the control laws, and generates actuator commands that are placed in a global COMMON section for access by TM on the next iteration. Its work finished, OBC hibernates, signaling SIMCON to wake up SIMOUT to write an analysis record to capture the results of this iteration. When SIMOUT hibernates, SIMCON wakes up TM to begin the next iteration.

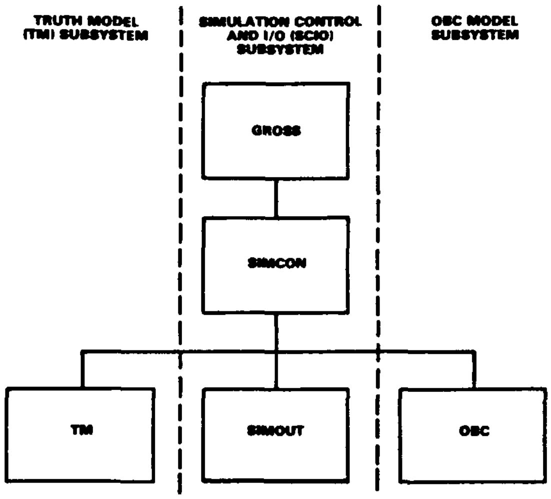

TRUTH MODEL (TM) SUBSYSTEM
SIMULATION CONTROL AND I/O (SCIO) SUBSYSTEM
OBC MODEL SUBSYSTEM
GROSS
SIMCON
TM
SIMOUT
OBC

Figure 6. FORTRAN Design: Hierarchy of Execution Tasks

The FORTRAN user can set the cycle time, which is the amount of time that the simulation clock is incremented. The cycle time determines when events occur in the simulation, for example, when thrusters fire, when new sensor data are generated, and when the spacecraft attitude state is updated. The FORTRAN design thus involves iterating over the three processes (TM, OBC, and SIMOUT), with the user-settable cycle time determining when events occur.

Figure 5 shows an excerpt from the Ada design corresponding to the simulator operation. The notation in the figure needs some explanation. The rounded rectangles are Ada packages. Circles denote off-page connectors, with the labels E1, E2, etc., referring to external files and the label 1 denoting package number 1 from a different diagram. Arcs show the direction of a subprogram call from a subprogram in the calling package to a subprogram in the called package. More detail on the design notation is presented in [10].

The placement of packages on design diagrams such as Figure 5 shows the seniority hierarchy described earlier. Thus, in Figure 5, the Simulation Control package is senior to other packages on the diagram; that is, it uses services provided by these other packages and they do not use its services. The three packages at the lowest level (which together constitute the simulation support subsystem of Figure 3) are junior to the packages higher in the diagram and as such are not the origin for any arcs that terminate at higher level packages.

This more detailed examination of the operation of each simulator revealed two clear differences in the Ada design: the passive role of the Truth Model and the separate timing of the OBC and the Truth Model.

The Ada design represented by Figure 5 shows that, unlike the FORTRAN design, the OBC and the Truth Model are not at the same level. The OBC calls the Truth Model to obtain sensor data when the data are needed. The Truth Model is passive; it performs processing and generates sensor data only when directed to do so.

Both the OBC and the Truth Model are junior to Simulation Control in Figure 5, an arrangement that appears to mimic the FORTRAN design. However, the Ada design notebook [11], which provides details of the actual calls made by Simulation Control, shows the Ada design to be quite different. Recall that the cycle time in FORTRAN affected both the OBC and the Truth Model. In the Ada design, the timing of the OBC and the Truth Model is separate: the Truth Model cycle time is under user control; OBC timing is not. The Ada team chose to model faithfully the spacecraft OBC flight software, whose timing is not under user control. Because timing and event scheduling are central elements in any simulation, this difference is of a fundamental nature and demonstrates that the Ada team was able to go back to basic system requirements for their analysis.

CONCLUSIONS

The comparison of FORTRAN and Ada designs has revealed significant differences in both the design processes and products. In this experiment, the Ada design has been shown to be different to a significant degree from the FORTRAN design. This result differs from that reported in [1] for another monitored Ada development project in a different environment.

The results have implications for other organizations contemplating the use of Ada. This experiment led to a design that exploits Ada's features for expressing design abstractions. However, this result was supported by (1) the use of a specification method, CSM, to counteract the influence of design-laden requirements; (2) the explicit allowance for the Ada team to pursue new design methods, not requiring the team to take the less costly route of reusing the existing design; and (3) training in alternative design methods.

ACKNOWLEDGMENTS

The Ada experiment is managed by F. McGarry and R. Nelson of NASA/GSFC and actively supported by representatives from all SEL participating organizations (NASA/GSFC, CSC, and the University of Maryland), especially V. Basili, G. Page, E. Katz, and C. Brophy. The authors thank J. Garrick, S. DeLong, G. Coon, D. Shank, and E. Seidewitz for their assistance.

REFERENCES

1. V. R. Basili et al., "Characterization of an Ada Software Development," Computer, September 1985, vol. 18, no. 9, pp. 53-65

2. Software Engineering Laboratory, SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

3. W. W. Agresti, "Measuring Ada as a Software Development Technology in the Software Engineering Laboratory (SEL)," Proceedings, Tenth Annual Software Engineering Workshop, NASA/GSFC, December 1985

4. Computer Sciences Corporation, CSC/SD-85/6106, Gamma Ray Observatory (GRO) Dynamics Simulator Requirements and Mathematical Specifications, G. Coon, April 1985

5. D. N. Card, V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986, vol. SE-12, no. 2, pp. 264-271

6. W. W. Agresti, "An Approach for Developing Specification Measures," Proceedings, Ninth Annual Software Engineering Workshop, NASA/GSFC, November 1984

7. Computer Sciences Corporation, CSC/TM-85/6108, Specification of the Gamma Ray Observatory (GRO) Dynamics Simulator in Ada (GRODY), W. W. Agresti, E. Brinker, P. Lo, et al., November 1985

8. V. Rajlich, "Paradigms for Design and Implementation in Ada," Communications of the ACM, July 1985, vol. 28, no. 7, pp. 718-727

9. E. W. Dijkstra, "The Structure of 'THE'-Multiprogramming System," Communications of the ACM, May 1968, vol. 11,, no. 5, pp. 341-346

10. E. Seidewitz and M. Stark, "Toward a General Object-Oriented Software Development Method," Proceedings, First International Symposium on Ada for the NASA Space Station, Houston, Texas, June 1986

11. Computer Sciences Corporation, CSC/SD-86/6013, GRO Dynamics Simulator in Ada (GRODY) Detailed Design Notebook, W. Agresti, E. Brinker, P. Lo, et al., March 1986

N89-16357

Max A. Turner
University of Houston-Clear Lake

P.D. Van Buskirk
Lockheed Engineering and Management Services Co

(PRELIMINARY) MODELING, SIMULATION AND CONTROL FOR A CRYOGENIC FLUID MANAGEMENT FACILITY

How relevant to the complex space station applications are academic formulation and solution of control problems, based on recently published textbook methodology complemented with limited laboratory scale experiments? The textbook abstractions are often stripped of consideration of constraints of prime concern to the field application: process capacities, user demands, economics, hazards analysis and fault tolerance. However, the approach of the classroom -- simplistic of necessity due to man-hour and funding constraints -- serves as a starting point for formulating a "top-down modular" definition of the problem and development of an overall perspective for the research professor or student. The individual is thus conditioned to readily adapt to a position in team efforts with major funding.

As one of an ongoing series of term projects in Process Monitoring and Control at UH-CL, the class in PROC 5232: Process Modeling, Simulation and Control, has studied the synthesis of a control system for a cryogenic fluid management facility. The severe demands for reliability as well as instrumentation and control unique to the space station environment are prime considerations.

Realizing that the effective control system depends heavily on quantitative description of the facility dynamics, a methodology for process identification and parameter estimation is postulated. A block diagram of the associated control system is also postulated. Finally, an on-line adaptive control strategy is developed utilizing optimization of the velocity form control parameters -- proportional gains, integration and derivative time constants -- in appropriate difference equations for direct digital control.

Of special concern are the communications, software and hardware supporting interaction between the ground and orbital systems. It is visualized that specialists in the OSI/ISO utilizing the Ada programming language will influence further development, testing and validation of the simplistic models here presented for adaptation to the actual flight environment.

MODELING, SIMULATION AND CONTROL
FOR
CRYOGENIC FLUID MANAGEMENT FACILITY

Abstract Outline

1. PROCESS DESCRIPTION

1.1 Baseline Configuration: CFMFE Flight System

The initial concept is diagrammed in Figure 1.1. Assembled as a module for mounting in the shuttle, it consists of three submodules identified with successive operational stages:

a) chilldown of the Ground Fill Line on the pad;

b) chilldown and filling of the Supply Tank on the pad;

c) chilldown of the Transfer Line combined with chilldown and filling of the Receiver Tank in orbit.

The submodules for operational stages a) and b) are detailed in Figure 1.1.1. The submodules for operational stages c) and d) are detailed in Figure 1.1.2.

NOTICE

At the deadline for submitting manuscripts this paper was incomplete.

Copies of the completed version will be made available at the presentation to those who desire one.

Max Turner
UH-Clear Lake Box 329
2700 Bay Area Blvd.
Houston, TX 77058-1098
(713) 488-9480

CFMF SIMPLIFIED SCHEMATIC

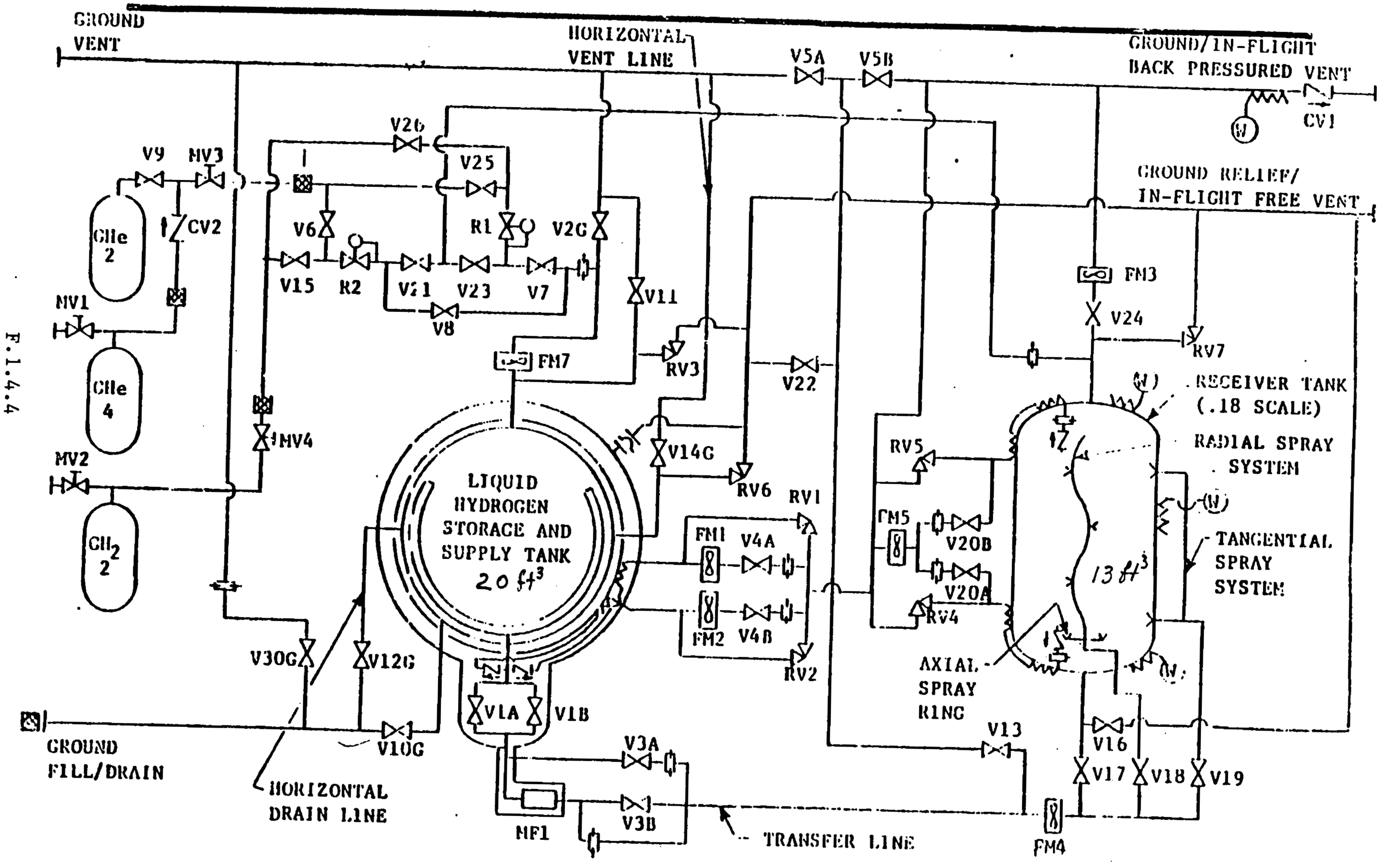


FIGURE 1.1 INITIAL CONCEPT

F.1.4.4

ORIGINAL PAGE IS OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

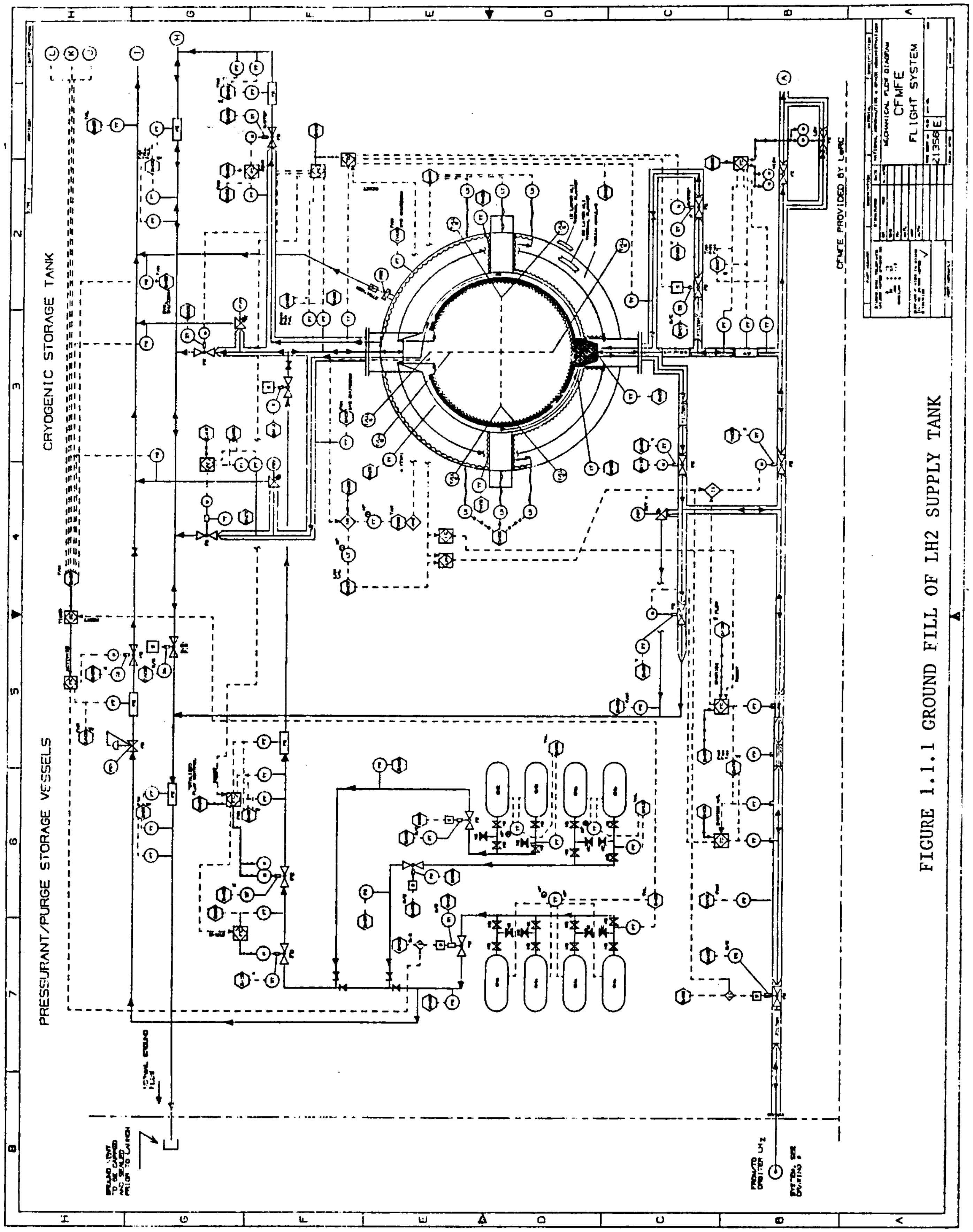


FIGURE 1.1.1 GROUND FILL OF LH2 SUPPLY TANK

ORIGINAL PAGE IS
OF POOR QUALITY

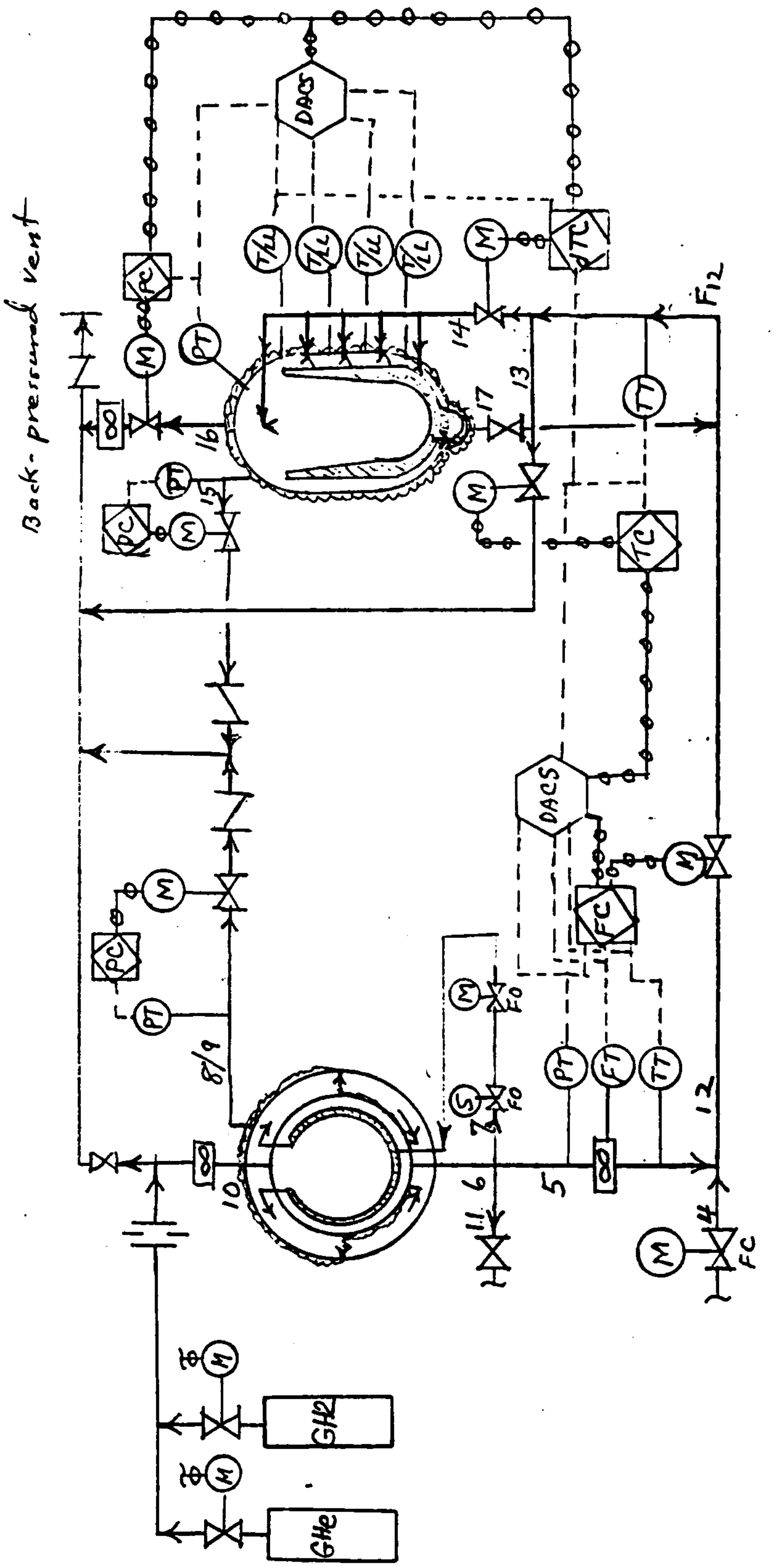


FIGURE 1.1.1 LH2 TRANSFER TO RECEIVER TANK

1. PROCESS DESCRIPTION

1.2 Functional Requirements and Constraints

Time:	Chilldown time,	$\theta_c = 15$ min.
	Fill time,	$\theta_F = 60$ min.
Pressure:	Minimum	$P_{min} = 1$ atm
	Maximum excursion	$P_{max} = 85$ psig (PSV spec)
Temperature:	Minimum	$T_{min} = 36.7^oR$ (20.4^oK)
	Ambient	$T_a = 530^oR$
	Maximum ΔT_{87}	$\Delta T_{87max} =$ (TBD)
Conservation of H2:	(TBD)	
Hazards:	Explosion and fire (TBD)	
	Destructive vibration (TBD) and shock	
	Stress fractures (TBD)	
	Loss of power (TBD)	
Zero-gravity:	Liquid pressurization (TBD)	
	Chilldown of receiver tank system (TBD)	
	Filling receiver tank (TBD)	
	Contingency respondent and fault tolerant (TBD)	

CFMFE

1. PROCESS DESCRIPTION

1.3 <u>Problem Identification</u>

1.3.1 Thermal balances and minimized system chilldown and fill times

On the pad:

1. The Ground Fill line
2. The CH2 Storage and Supply Tank

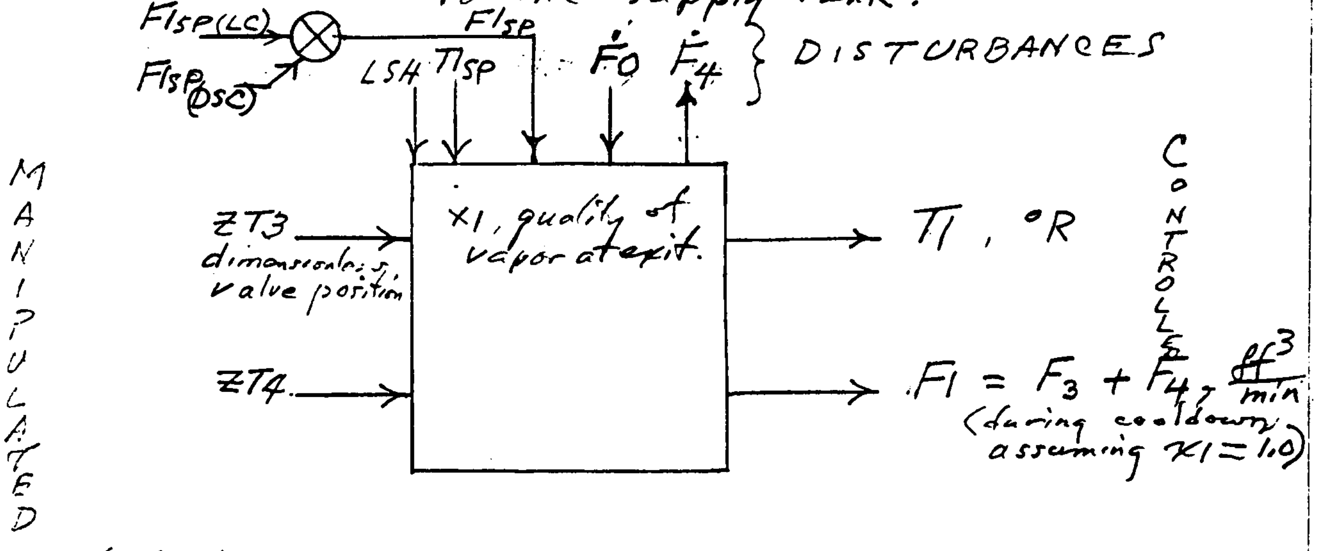
In orbit:

3. Chilldown time for the transfer line from the Supply Tank to the Receiver Tank
4. Chilldown and fill time for the Receiver Tank

Special problems:

5. Overpressures and destructive stresses
6. Delayed GH2 boiloff due to heat transfer

The I/O model, for the transfer line from the LH2 source to the supply tank:



disturbances:

$Tl_{SP(t)} = A_T + B_T t_D$, °R (TBD) (dictated by computer)

where $t_D = t/\theta$
θ = cooldown time, i.e., the time until $T_1 - T_{LH2} < D_{lmax} \cong 5°K$ (?)
LC = average measured LH2 level in supply tank.

$Fl_{SP(t)} = Fl(LC, TS)$, ft³/min (TBD) (during cooldown $x1 = 1.0$; when filling the supply tank, $x1$ = small fraction or negligible) (dictated by computer)

$\dot{F}0 = \dot{F}0$ (input LH2 flowrate defined by the fuel cell facility) (TBD)

$\dot{F}_4 = \dot{F}_4$ (demanded by the follow-on supply tank module for cooldown and filling period) (TBD)

LSH

controlled variables:

Tl = line 1 temperature subject to constraints $|dT_1/dt| < D_{max}$, $Pl_{Low} < Pl < Pl_{HIGH}$ (TBD)

Fl = line 1 volumetric flowrate controlled mainly to limit maximum possible pressure surges in the line (TBD)

manipulated variables

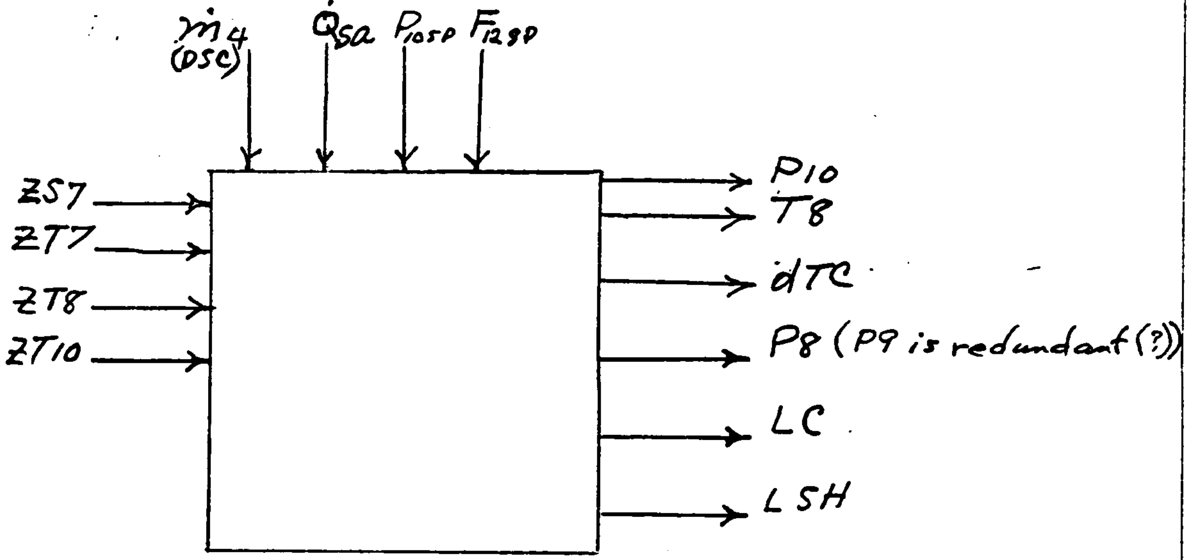
$ZT3 = PI(Tl)$

$ZT4 = PI(Fl)$

$ZS1 = DDC \cdot \overline{LSH}$ (logical)

Problems: (omit)
1.2 Draw the appropriate block diagram analogous to Fig. 29.10 in Stephanopoulos.
1.3 Write the closed-loop transfer functions.

on the pad

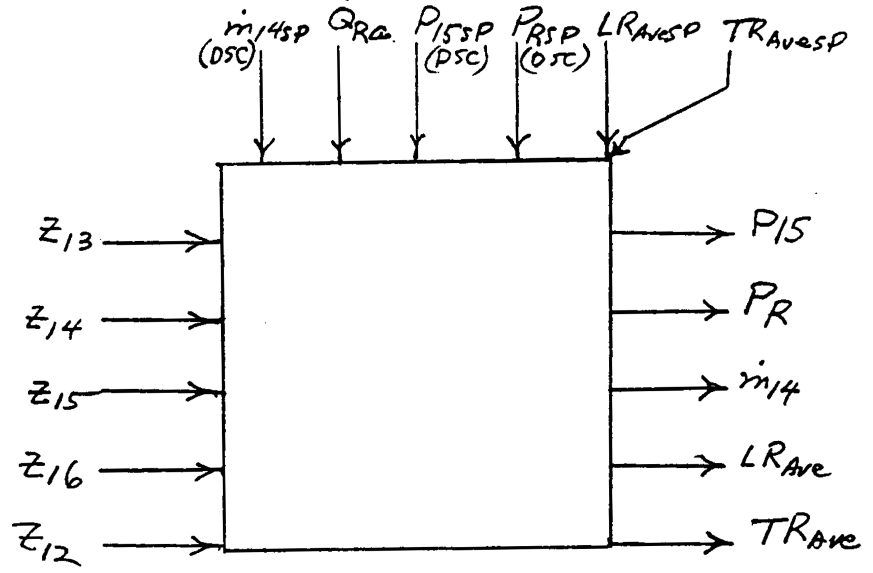


disturbance: $\dot m_4 = \dot m_4(F1, ZT3, ZT4, T1, P1, \dot Q_1)$ (TBD)
$\dot Q_a$ = nil for loading but contributes to boiloff afterwards (TBD)

controlled:
P10 ≈ Supply tank pressure, $P10 < P_{10max}$ (TBD)
T8 = Cooling coil vent gas temperature
dTC ≡ $T8 - T7 < D_{max}$ (TBD)
P8 = cooling coil vent gas pressure $< P8_{max}$ "
LC = average of 4 supply tank measurement
LSH = supply tank level high causing interrupt closing of ZT4

manipulated
ZS7 = $(dTC < dTC_{max})$ (TBD)
ZT7 = PID$(dTC = T_8 - T_7)$ "
ZT8 = PI (P8) "
ZT10 = PI (P10) "

in orbit



disturbances: $\dot{Q}_{Ra} = \dot{Q}_{Ra}$ (energy stored in concentrated masses, heat leakage from environment)
$\dot{m}_{14SP} = \dot{m}_{14SP}$ (supervisory control by DACS)
$P_{15SP}, P_{RSP}, L_{AveSP}, TR_{AveSP}$ = fcns(time, supervisory control by the DACS)

controlled: P_{15} = cooling coil vent gas pressure, $P_{15} < P_{15max}$ (TBD)
P_R = receiver tank pressure, $P_R < P_{Rmax}$
$\dot{m}_{14}$ = H2 mass flow rate, a fcn of time responding to the setpoint scheduled by the DACS.
LR_{Ave} = average of liquid level measurements in tank, a fcn of time responding to the set point scheduled by the DACS
TR_{Ave} = average of tank temperature measurements, a fcn of time responding to the set point scheduled by the DACS.

manipulated: $Z_{12} = PI(F5)$
$Z_{13} = PI(T12)$
$Z_{14} = PID(\delta TC_{TR,13})$
$Z_{15} = P(P_{15})$
$Z_{16} = P(P_R)$

2. MATHEMATICAL MODELING

2.2 Analysis, Degrees of Freedom and Control Loops

2.2.1 The Ground Fill Line

Physical model for an energy balance:
Assumptions:

1. The aluminum tube is perfectly insulated
2. LH2 enters with quality $x = 0$
3. Until childown is essentially complete, the exit GH2 has a quality of $x = 1$
4. Maximum childown rates are limited by the venting capacity of the line
5. Significant thermal energy sources which limit the minimum cooldown time are the concentrated masses associated with stainless steel control valves and sensors.
6. The enthalpy of LH2 at near atmospheric pressure is given by:

$$h = 278.4 + 441.8x + 10.13\ (T-21)$$

$$= 507.47 + 10.13T,\ \text{kJ/kg} \quad \text{using the unit } {}^{o}K$$

$$= 218.63 + 2.425T,\ \text{Btu/lb}_m \quad \text{using the unit } {}^{o}R$$

Reference: Perry and Green, Ch.E. Handbook, McGraw-Hill 1984, pp 3-1958

$$C_{pf} = C_{pg} = 10.13\ \text{kJ/kg}^{o}K \text{ at } 21^{o}K$$

7. The heat capacity of Al is:

$$C_{vAl} = -\ 0.1362 + 0.007528T - 0.00001356T^2 \quad \text{kJ/kg}^{o}K$$

with T in ^{o}K

$$C_{vAl} = -\ 0.03254 + 0.000999T - 9.99 \times 10^{-7}T^2 \quad \text{Btu/lb}_m{}^{o}R$$

with T in ^{o}R

Reference: Perry and Green, 1984, pp 3-135

8. The heat capacity of stainless steel is:

$$C_v = -\ 0.0586 + 0.003219T - 5.078 \times 10^{-6}T^2 \quad \text{kJ/kg}^{o}K$$

with T in ^{o}K

$$C_v = -\ 0.0140 + 0.000428T - 3.75 \times 10^{-7}T^2 \quad \text{Btu/lb}_m{}^{o}R$$

with T in ^{o}R using $1\ \text{Btu/lb}_m{}^{o}R = 4.178\ \text{kJ/kg}^{o}K$

2.2.1 The Ground Fill Line

Assumptions (continued)

9. Radiation heat transfer rates across the annulus of concentric tubes or spheres is nil compared to convective heat transfer rates from Al to LH2:

$q/A \overset{0}{=} 300$ Btu/hrft^2 from Al to LH2 at 36.7^oR

$q/A = F_{12}\sigma(T^4 - T^4_{LH2}) \overset{0}{=} 12.7$ Btu/ft^{2}hr from StSt to Al at 530^oR where $F_{12} = \frac{1}{\frac{1}{\epsilon_1} + \frac{1}{\epsilon_2} - 1} \overset{0}{=} \frac{1}{\frac{1}{.1} + \frac{1}{.2} - 1} \overset{0}{\simeq} 0.094$

$\sigma = .1713 \times 10^{-8}$ Btu/(ft^{2}hr^{o}R^4)

Reference: Perry and Green, 1984

10. Thermal diffusivities, $k/\rho C_v$:

	Al	St.St.
20^{o}K	0.5	0.040
100^{o}K	0.00023	$\sim 9 \times 10^{-5}$
300^{o}K	0.00011	3.3×10^{-5}

Reference: Perry and Green, 1984, pp 3-263

11. ρ_{GH2} = 0.104 kg/m^3 at 20.4^{o}K

ρ_{LH2} = 70.57 kg/m^3 at 20.4^{o}K or 4.72 16_m/ft^3

11. Densities: ρ_{Al} = 2723 kg/m^3 or 170 lb$_m$/ft^3

ρ_{StSt} = 7900 kg/m^3 or 492 lb$_m$/ft^3

Reference: Perry and Green, 1984, pp 3-96

12. Thermal conductivities:

Reference: Perry and Green, 1984, pp 3-261

13. Convective heat transfer coefficients:

References:
1) Perry and Green, 1984, pp 10-23
2) H.H. Walters, AiResearch Manufacturing Compant "Single-Tube Heat Transfer Tests with Liquid Hydrogen", (see WADC Technical Report 59-423)
3) Drake et al., Arthur D. Little, Inc. "Pressurized Cool-Down of a Cryogenic Liquid Transfer system Containing Vertical Sections", (tests with LO2)

2.2.1 The Ground Fill Line

Assumptions (continued)

Walters -- LH2 tests:

film boiling:	$h = 460$ to 540 Btu/hrft^{2o}R for inlet (?) = 1.6 to 1.7 atm $Re \overset{0}{=} 3 \times 10^5$
nucleate boiling:	h = 10 x value for film boiling

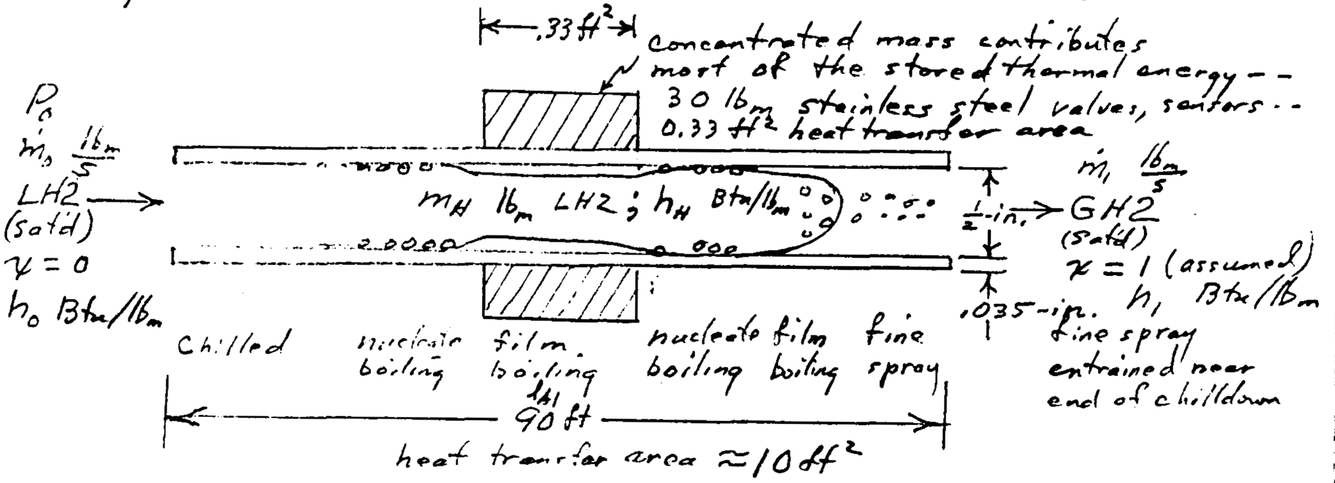
Drake et al. -- LO2 tests:

film boiling:	$h = 300$ Btu/ft^{2}hr^{o}R for inlet pressure = 20 psig outlet pressure = 10 psig $h = 200$ Btu/ft^{2}hr^{o}F for inlet pressure = 10 psig outlet pressure = 5.5 psig
LH2 -- assume:	$h_{max} = 500$ Btu/hrft^{2o}R (uncontrolled) $h_{ave} = 300$ Btu/hrft^{2o}R (controlled) $Re \overset{0}{=} 3 \times 10^5$

14. Critical constants of H2: $P_c = 12.8$ atm

$T_c = 33.3^oK = 60.0^oR$

<u>physical description</u> (nominal dimensions and masses)


.33 ft²
concentrated mass contributes most of the stored thermal energy -- 30 lbm stainless steel valves, sensors -- 0.33 ft² heat transfer area
P_0
$\dot{m}_0$ lbm/s
LH2 (sat'd)
$x = 0$
h_0 Btu/lbm
m_H lbm LH2; h_H Btu/lbm
$\dot{m}_1$ lbm/s
$\frac{1}{2}$-in.
GH2 (sat'd)
$x = 1$ (assumed)
h_1 Btu/lbm
.035-in.
Chilled
nucleate boiling
film boiling
nucleate boiling
film boiling
fine spray
fine spray entrained near end of chilldown
90 ft
heat transfer area ≈ 10 ft²

thermal balances and heat transfer problems

Initially, the Ground Fill Line is at ambient temperature. As LH2 is introduced the liquid phase moves through the line in such a way that the internal Al tubing is chilled within a short distance behind the advancing LH2 front. Prolonged boiling occurs over contact surfaces associated with the concentrated masses.

If the LH2 is introduced at an adequate rate, the maximum cooldown rate at near atmospheric outlet pressure(P_1) is limited by the venting capacity of the tube at the given inlet pressure (P_0)

If the LH2 is introduced at an excessive rate, explosive overpressures and thermal contraction can result in destructive stresses.

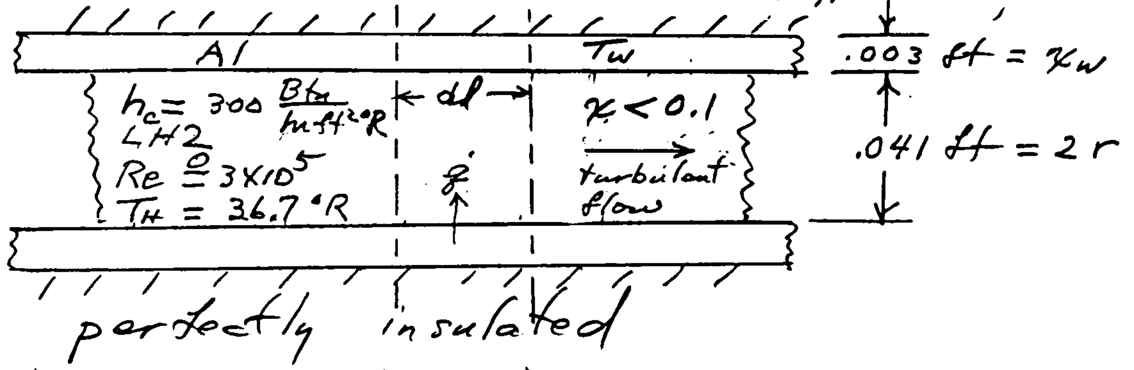
Four problems are addressed:

Prob. 1. Cooldown time for a perfectly insulated infinitely long Al tube at ambient temperature suddenly filled with LH2 at atmospheric pressure;
Prob. 3. cooldown time for a finite concentrated mass joined to the Al tubing over a finite area;
Prob. 2, potential overpressures during the cooldown of the infinitely long Al tube suddenly filled with LH2; and
Prob. 4. minimum cooldown time dictated by the venting capacity of a finite tube to which concentrated masses are joined.

(A hypothetical case not physically possible.)

Problem 1: Cooldown time for a perfectly insulated infinitely long Al tube at ambient temperature suddenly filled with LH2 at constant pressure of 1 atmosphere, and with $Re \overset{o}{=} 3 \times 10^5$.

ORIGINAL PAGE IS OF POOR QUALITY



perfectly insulated

$$\dot{q} = 2\pi r\, dl\, h_c (T_W - T_H)$$

$$= -\rho_W 2\pi r\, x_W\, dl\, c_W \frac{dT_W}{dt}$$

$$= \pi r^2 dl\, \rho_H\, h_{fg} \frac{dx}{dt}\ ;\ Btu/hr$$

where heat of vaporization, $h_{fg} = 441.8\ kJ/kg$
$dT_H/dt = 0$ (assumed) $= 190\ Btu/lbm$
density, $\rho = 170\ lbm/ft^3$

$$c_W \approx -0.014 + 0.000428\,T - 3.75\times10^{-7}\,T^2 \quad Btu/lbm\text{-}°R$$

dividing by $2\pi r\, dl\, h$ $= 0.0509$ Btu/lbm°R at 180°R
$= 0.0915$ " at 360°R

$$T_W - T_H = -(\rho_W\, x_W\, c_W/h)\frac{dT_W}{dt} = -\tau_W \frac{dT_W}{dt}$$

$$= \frac{1}{2} r\, \rho_H \frac{h_{fg}}{h_c}\frac{dx}{dt}$$

assuming T_H is constant and heat transfer to LH2 merely converts the liquid to vapor

$$\tau_W \frac{dT_W}{dt} + T_W = T_H$$

define $D_W = T_W - T_H$, $\frac{dT_W}{dt} = \frac{dD_W}{dt}$

$$\tau_W \frac{dD_W}{dt} + D_W = 0$$

$$D_W(t) = D_{W(0)}\ e^{-t/\tau_W}$$

(later: integrate for $c_W = c_W(t)$)

First approximation

$$T_{W(t)} = (T_{W(0)} - 36.7)\exp\{-t/[(170 \times .003 \times .09/500)\cdot 3600]\} + 36.7\ °R$$

$$T_{W(t)} = (T_{W(0)} - 36.7)\, e^{-2.977t} + 36.7\ ,\ °R$$

with time in seconds

F.1.4.16

problem 1 (cont'd):
For $T_{W(0)} = 536.7\,^{\circ}R\ (298.2\,^{\circ}K)$

$$T_{W(t)} = 500\exp(-2.977t) + 36.7,\quad ^{\circ}R$$

t, sec	$T_{W(t)}$, °R	
0	536.7	
1	62.2	
2	38.0	(1.3°R above b.pt.)
3	36.77	(0.07°R above b.pt.)

} nucleate boiling reduces these values for $T_W - T_H < 5\,^{\circ}R$

problem 2: the potential overpressure during cooldown of the infinitely long Al tube of problem 1 suddenly filled with LH2

if assume $x_{(0)} = 0$, i.e., the line is completely filled with LH2; and, further, if assume absolutely no strain in the tube wall as well as zero compressibility of the LH2, then all heat transfer yields an increase in enthalpy (h_{LH2}) of the LH2:

$$2\pi r\, x_W\, d\ell\, \rho_{AL}\int_{T_a}^{T_{LH2},P_{atm}} c_{AL}\,dT \overset{=}{=} \pi r^{2}\,d\ell\,\rho_{LH2}\int_{T_{LH2},1atm}^{T_{LH2},P_{atm}} c_{pf}\,dT$$

$$2\times.003\times170\int_{T_a}^{T_{LH2},P_{atm}}\left[-0.03254 + 0.000999T - 9.99\times10^{-7}T^{2}\right]dT$$
$$= .041\times4.72\int_{T_{LH2},1atm}^{T_{LH2},P_{atm}} 2.425\,dT$$

$$-0.0325(T_{LH2,P_{atm}} - 530) + \underline{0.000999}_{\,2}(T^{2}_{LH2,P_{atm}} - 530^{2})$$
$$-3.33\times10^{-7}(T^{3}_{LH2,P_{atm}} - 530^{3}) \overset{=}{=}$$
$$\frac{.041\times4.72\times2.425}{2\times.003\times170}(T_{LH2,P_{atm}} - 36.7)$$

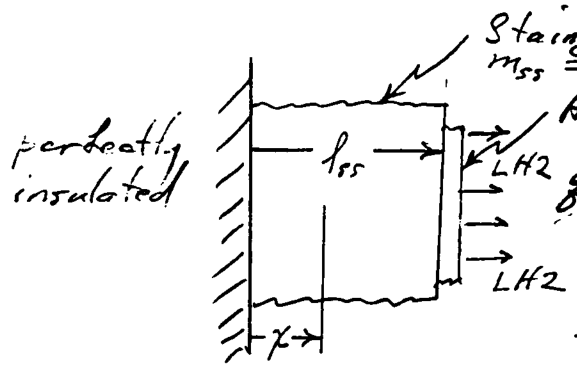
$$T_{LH2,P_{atm}} \approx 185\,^{\circ}R \quad (\text{supercritical})$$

$$P_{LH2} \overset{\circ}{\simeq} \frac{\rho RT}{144} = 4.72\times766\times185/144$$
$$= 4650\ \text{lbf/in}^{2}$$

potential overpressure $\overset{\circ}{=}$ 300 atm.
within 3 seconds (?)

problem 3: Cooldown time for a concentrated mass:

Reference: J.C. Burke et al, Arthur D. Little, Inc. "Pressurized cooldown of Cryogenic ~~Transfer~~ Lines" F-5, (?) -- based on tests with $LN2$.


Stainless Steel slab with mass $m_{ss} \overset{\circ}{=} 30\ lb_m$
Al tube wall
perfectly insulated
l_{ss}
LH2
LH2
x

$$g = hA_c(T_{(l_{ss},t)} - T_{LH2})$$
$$\overset{\circ}{=} 300 \times .33(T_{(l_{ss},t)} - 36.7)\ \frac{Btu}{hr}$$
$$l_{ss} \approx \frac{30}{492} \times \frac{1}{.33} \overset{\circ}{=} 0.185\ ft.$$
(2.2-in)

assume: $\frac{dT}{dx} = 0$ in the Al tube wall
$P = 1\ atm$

By Fourier field equation

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2} \quad \frac{°R}{hr} \quad \text{within the steel slab (a)}$$

where $\alpha = \frac{k}{\rho C_v} \approx 8 \times 10^{-5} \frac{m^2}{s}$

initial & boundary conditions

$t = 0 \quad T_{(x,0)} = T_a = 530°R$

$x = 0 \quad \frac{\partial T(0,t)}{\partial x} = 0$

$x = l_{ss} \quad g = hA_c(T_{(l_{ss},t)} - T_{LH2}) = kA_c \frac{\partial T_{(l_{ss},t)}}{\partial x}$

in dimensionless form

$\Theta \equiv$ cooldown time, i.e., the time required to cool the slab to negligible $\Delta T = T_{ave} - T_{LH2} \leq E_{\Delta T}$ (TBD) (?)

$t_D = t/\Theta \quad 0 < t_D < 1.$

$k_D = \frac{k}{l_{ss}} \longrightarrow \alpha_D = \frac{\alpha_{ss}\Theta}{l_{ss}^2}$

$x_D = x/l_{ss} \quad 0 < x_D < 1$

$$\boxed{\begin{aligned} &\frac{\partial T}{\partial t_D} = \alpha_D \frac{\partial^2 T}{\partial x_D^2} \qquad \text{(b)} \\ &t = 0, \quad T_{(x_D,0)} = T_a \\ &x_D = 0, \quad \frac{dT_{(0,t_D)}}{dx_D} = 0 \\ &x_D = 1, \quad \frac{\partial T_{(1,t_D)}}{\partial x_D} = \frac{h}{k_D}(T_{(1,t_D)} - T_{LH2}) \end{aligned}}$$

note: $\frac{h}{k_D} \overset{\circ}{\approx} \frac{.500}{.0566/.185} \overset{\circ}{\approx} 1634$

where $k_{S1S1} = 0.0566 \frac{Btu}{ft\,hr\,^\circ R}$ at 200 R

$\therefore$ assume

$$\frac{h}{k_D} = \infty$$

$$\frac{\partial T(1,t_D)}{\partial x_D} = \infty$$

Laplace domain

$$s\,T_{(x_D,s)} - T_{(x_D,0)} = \alpha_D \frac{\partial^2}{\partial x^2} T_{(x_D,s)}$$

$$\frac{\partial^2}{\partial x^2} T_{(x_D,s)} - \frac{s}{\alpha_D} T_{(x_D,s)} = -\frac{1}{\alpha_D} T_a$$

which yields a general solution in the Laplace domain as a function of x_D

$$T_{(x_D,s)} = A_{(s)}\, e^{\sqrt{\frac{s}{\alpha_D}}\,x_D} + B_{(s)}\, e^{-\sqrt{\frac{s}{\alpha_D}}\,x_D} - \frac{T_a}{\alpha_D}$$

$$T(0,s) = A_{(s)} + B_{(s)} - T_a/\alpha_D$$

$$B_{(s)} = T_{(0,s)} + T_a/\alpha_D - A_{(s)}$$

$$T(1,s) = A(s)\, e^{\sqrt{\frac{s}{\alpha_D}}} + \left(T_{(0,s)} + \frac{T_a}{\alpha_D} - A_{(s)}\right) e^{-\sqrt{\frac{s}{\alpha_D}}} - \frac{T_a}{\alpha_D} \approx \frac{T_{LN2}}{s} \qquad (C)$$

since by the boundary condition at $x_D = 1$:

$$\frac{\partial}{\partial x_D} T_{(1,s)} = \frac{h}{k_D}\left(T_{(1,s)} - \frac{T_{LN2}}{s}\right)$$

$$T_{(1,s)} = \frac{T_{LN2}}{s} + C\,e^{-\frac{h}{k_d}} \quad \text{where } C\,e^{-1634} \approx C\,e^{-\infty}$$

$$= \frac{T_{LN2}}{s}$$

$A(s)$ = (explicit relationship from eq (C)) which can be expressed as a series relationship in s.

(TB Completed Later)

Solution of eq.(6) by separation of variables

$$T = \Theta(t)\,\Psi(x)$$

$$\frac{\partial T}{\partial t_D} = \Psi(x)\frac{\partial \Theta}{\partial t_D}, \quad \frac{\partial^2 T}{\partial x_D^2} = \Theta(t)\frac{\partial^2 \Psi(x)}{\partial x_D^2}$$

$$\Psi\frac{\partial \Theta}{\partial t_D} = \alpha_D\,\Theta\frac{\partial^2 \Psi}{\partial x_D^2}$$

$$\frac{1}{\Theta}\frac{\partial \Theta}{\partial t_D} = \alpha_D\frac{1}{\Psi}\frac{\partial^2 \Psi}{\partial x_D^2} = -H$$

(a positive H yields a solution that increases without limit as $t \to \infty$)

$$\Theta = C_0 e^{-Ht_D}$$

$$\Psi = C_1 e^{j\sqrt{\frac{H}{\alpha_D}}x_D} + C_2 e^{-j\sqrt{\frac{H}{\alpha_D}}x_D}$$

$$T = e^{-Ht_D}\left(C_1 e^{j\sqrt{\frac{H}{\alpha_D}}x_D} + C_2 e^{-j\sqrt{\frac{H}{\alpha_D}}x_D}\right)$$

$$\frac{\partial T}{\partial x_D} = e^{-Ht_D}\,j\sqrt{\frac{H}{\alpha_D}}\left(C_1 e^{j\sqrt{\frac{H}{\alpha_D}}x_D} - C_2 e^{-j\sqrt{\frac{H}{\alpha_D}}x_D}\right)$$

$$\left.\frac{\partial T}{\partial x_D}\right|_{x_D=0} = e^{-Ht}\,j\sqrt{\frac{H}{\alpha_D}}\left(C_1 - C_2\right) = 0$$

$$C_2 = +C_1$$

$$T = e^{-Ht_D}\,2C_1\cos\sqrt{\frac{H}{\alpha_D}}x_D$$, a particular soln which leads to a general solution

$$T = \sum_{n=0}^{\infty} e^{-H_n t_D} K_n \cos\left(\sqrt{\frac{H}{\alpha_D}}x_D + 2n\pi\right) \quad (7)$$

(To be completed later)

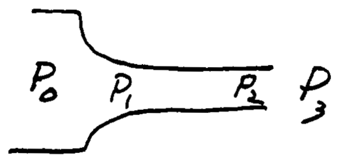
Problem 4. minimum cooldown time dictated by the venting capacity of a finite tube to which concentrated masses are attached

at the tube exit assume;

sonic velocity for saturated GH2 at 1 atmosphere

for $k = 1.6$
the limiting ratio

$$\frac{P_2}{P_0} = \frac{P_1}{P_0} = 0.497$$



and the maximum mass velocity for isentropic flow is

$$G^\circ = P_0\sqrt{\frac{g_c M k}{R T_0}\left(\frac{2}{k+1}\right)^{(k+1)/(k-1)}} \quad, \text{kg/s}\cdot\text{m}^2$$

$$= 1.01\times10^{6}\sqrt{\frac{1\times2\times1.6}{8314\times36.7}\left(\frac{2}{1.6+1}\right)^{(1.6+1)/(1.6-1)}}$$

$$= 583 \text{ kg/s m}^2$$

$$= 119 \text{ lb}_m/\text{s}\cdot\text{ft}^2$$

$$G^\circ A_{tube} = 119 \times \frac{\pi}{4} \times \left(\frac{.5}{12}\right)^2$$

$$= 0.163 \text{ lb}_m \text{ GH2}/\text{s}$$

lb$_m$ LH2 required to remove sensible heat from Ground Fill Line

$$\Big[30 \text{ lb}_m \text{ SST} \times .09 \frac{\text{Btu}}{\text{lb}_m \text{SST}^\circ\text{R}} + 7 \text{ lb}_m \text{Al} \times .19 \frac{\text{Btu}}{\text{lb}_m\cdot\text{Al}^\circ\text{R}}\Big] \times (536.7 - 36.7)^\circ\text{R}$$

$$= \qquad \text{Btu}$$

$$\text{cooldown time} \geq \frac{2017 \text{ Btu}}{0.163 \frac{\text{lb}_m\text{GH2}}{\text{s}} \times (4418/1.055\times2.2)\frac{\text{Btu}}{\text{lb}_m\text{GH2}}}$$

$$\geq 65 \text{ seconds}$$

twophase flow rate through 90 ft. of tubing will entail significant frictional losses!
(T B Completed later)

Background: The previous presentation for the Transfer Line
Perry & Green, Ch.E. Handbook, p10-59:
eq.(10-195) where $A_1\mathcal{F}_{12} \equiv A_2\mathcal{F}_{21}$

$\sigma = (0.1713)(10^{-8})\frac{Btu}{ft^2 \cdot h \cdot {}^\circ R^4}$ $= \frac{1}{\frac{1}{A_1\epsilon_1} + \frac{1}{A_2}\left(\frac{1}{\epsilon_2} - 1\right)}$

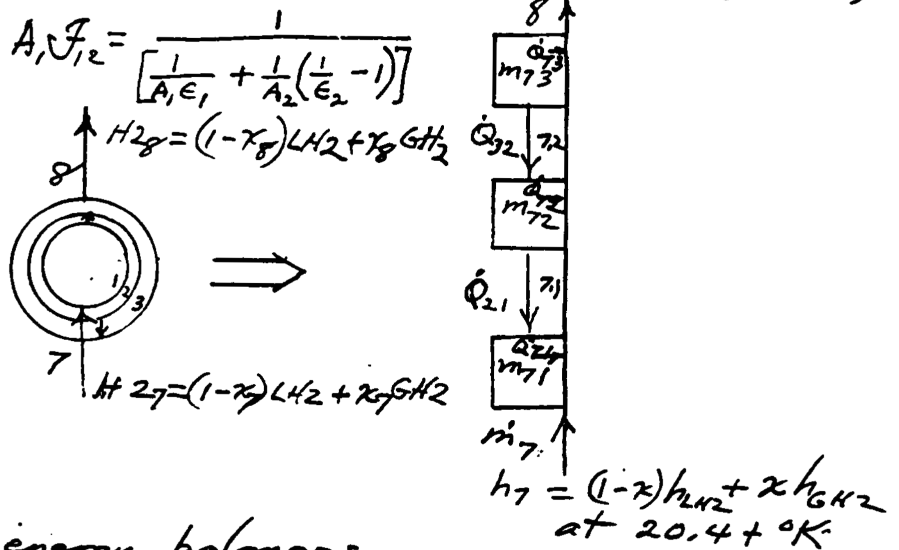
$\epsilon_1 \approx 0.10$ for Al

$\epsilon_2 \approx 0.60$ for St-Steel

$$\dot{Q}_{i\to j} = A_i\mathcal{F}_{ij}\sigma T_i^4 - A_j\mathcal{F}_{ji}\sigma T_j^4$$
$$= A_1\mathcal{F}_{12}\sigma(T_1^4 - T_2^4)$$
$$= A_1\mathcal{F}_{12}\sigma(T_1^3 + T_1^2T_2 + T_1T_2^2 + T_2^3)(T_1 - T_2)$$

FIGURE 2.1

$$A_1\mathcal{F}_{12} = \frac{1}{\left[\frac{1}{A_1\epsilon_1} + \frac{1}{A_2}\left(\frac{1}{\epsilon_2} - 1\right)\right]}$$



Problems:
1.5 Write the energy balances over submasses accounting for transfer of heat between submasses as well as to the H2 coolant.

ORIGINAL PAGE IS OF POOR QUALITY

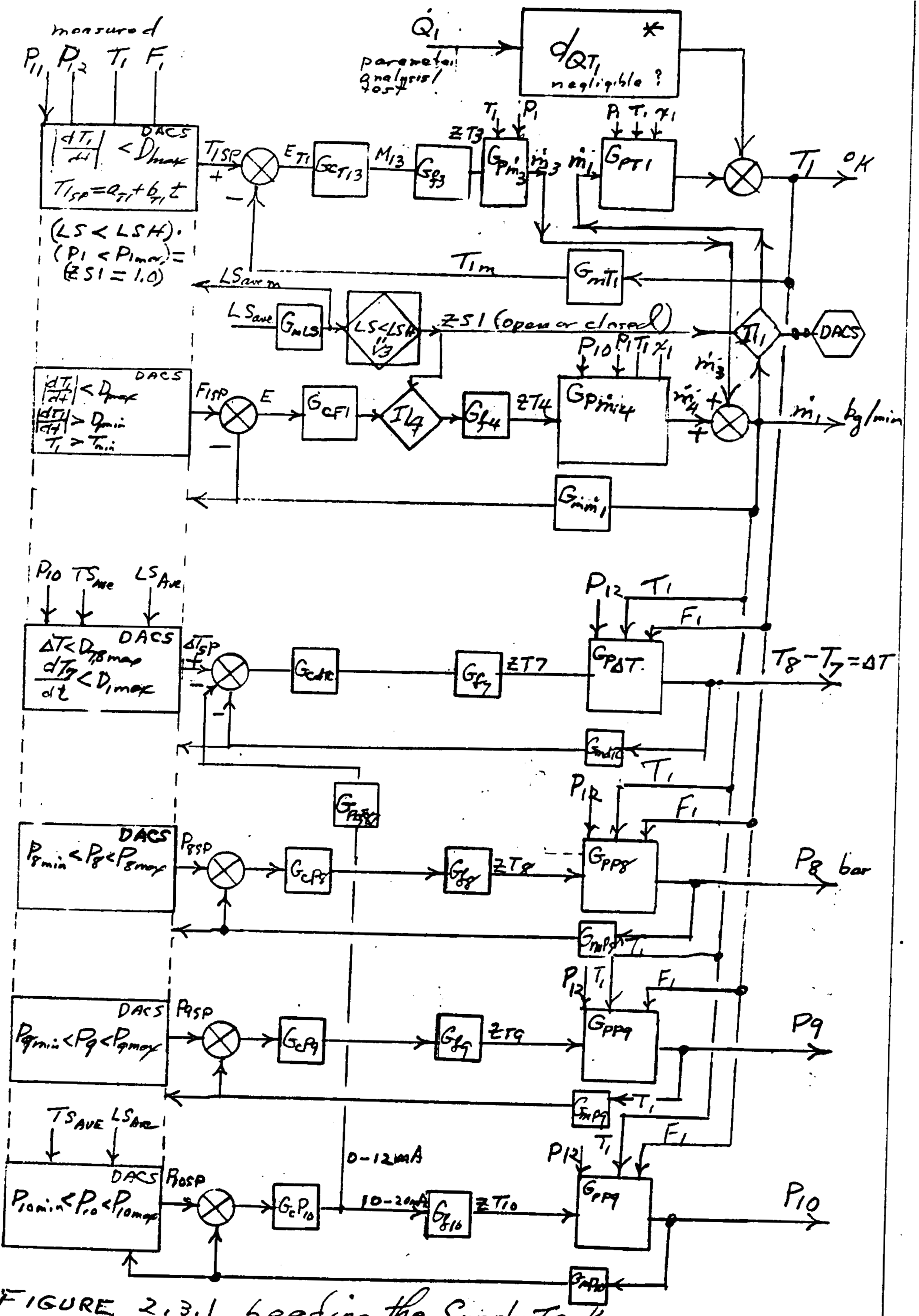


FIGURE 2.3.1 Loading the Supply Tank

* Note: With good thermal design, heat leakage into the system is negligible during the cooldown and loading process. It becomes the primary concern when steady state boiloff occurs after loading is complete. There remains heat exchange between components 8, 9 & 10.

1 bar ≡ 100 kN/m² (kPa) 0.987 atm.

Intertask Communication in Ada:
A Bus Interface Solution

This is a special two hour presentation by representatives of the Weizmann Institute Rehovoi, Israel. The topics are listed below:

Statement of the problem: Intertask synchronization in real time systems.
Hierarchical Partitions - a simpler reference model
Data Link Layer Regime: The prioritized dialogue with provision for message cancellation.
Message Alphabets
Ada Oriented Protocols and extension to CSP.
VLSI Solution

Flavia Rosenberg
Smil Ruhman
A. Pnueli

N89-16358

VERIFYING PERFORMANCE REQUIREMENTS

By
Dr. Joseph Cross
Sperry Corporation
St. Paul, MN
(612) 456-7316

INTRODUCTION

The thesis presented in this paper is that today, it is in general impossible to verify that the performance requirements on a software program will be met. An approach to a partial solution to this problem is presented.

The next section of this paper, Problem Definition, defines the problem to be addressed, and defines related terms as they are used below.

The following section, Obstacles to Verifying Performance Requirements, presents the reasons why performance requirements are, today, difficult to verify.

The section on Methods for Verifying Performance Requirements briefly presents methods in use today, and proposes an alternative approach to overcome some of the remaining difficulties.

PROBLEM DEFINITION

A "performance requirement" is a requirement on the speed of a function performed by software. Much of the following applies equally well to requirements on the amount of memory used by a software function. An example of a performance requirement is "The interval between updates to each track shall be on the average at most two seconds, and in no case longer than five seconds." Note that while performance requirements are, at the user level, generally stated in elapsed time, these requirements may be recast at lower levels of design into units of processor utilization.

"Verifying" a specific requirement on a specific software development work product refers to determining whether that requirement is fulfilled by that work product. The requirements on the work products of each phase of software development are results of the preceding phase, except for the system requirements, which are input to the entire software development process. A work product WP is said to satisfy a requirement R if <u>any</u> system produced according to the requirements set forth in WP (and its sibling work products, if any) will meet the requirement R. Verifying a software development work product in its entirety also entails checking its completeness, consistency, feasibility, and testability [1].

For example, to verify that a detailed design satisfies a requirement, such as the example requirement above, is to determine whether any system produced in accord with that detailed design could fail to exhibit the required behavior. Moreover, verifying the entire detailed design requires determining whether there is at least one system that can be built in accord with that detailed design.

Of course, what work products are produced and what are the phases of software development depend on the approach to software development in use. In the conventional approach, the phases are requirements analysis, design (often subdivided into high-level design and detailed design), and implementation; the work products of which are a requirements specification, a design document (or documents), and code, respectively. In the operational approach to software development, the first phase produces a prototype/executable specification, which is intended to satisfy/define all requirements except performance requirements; then a second phase transforms that prototype/executable specification into a program with the same behavior except that the performance requirements are met [2].

In order to minimize the dependence of the following discussion on the approach to software development in use, it will be assumed below that the work product on which performance requirements are to be verified is a body of compiled but untested Ada (tm) code. (Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.) This body of code could represent a detailed design in the conventional approach, or an intermediate step in the transformation of a prototype/executable specification in the operational approach.

In order to make the issues involved in the verification of performance requirements as simple as possible, it will be assumed that a target machine is given and fixed throughout the discussion. Here "target machine" refers to the virtual machine on which compiled code is to run: one or more processors, memories, communication channels, together with run-time support software such as operating systems. This target machine is assumed to be the target of a valid implementation of the Ada language.

Note that while the assumption of a single, known, target machine is reasonable in the Space Station environment, it is not reasonable in other environments in which the target machines that will execute the software may be unknown. We are fortunate in this regard.

The term "mapping" will be used to refer to the association between design-level objects and run-time objects. For example, a subprogram may be mapped onto a segment of memory-resident machine code, or it may be mapped into many similar segments of machine code (as it would be if it were inlined), or it may be mapped into nothing (as may be the case for type conversion functions). As another example, a data object may be mapped into a location in the main memory of one computer, into a register of one computer, or into several locations in several computers (as would be the case if redundant data were being maintained).

The function of mapping a program is generally distributed among the compiler, linker, loader, and run-time system.

Note that one of the goals of the Ada language design was to include in the source code all details of the program that define its semantics, <u>except for performance issues</u>. That is, by examination of only the source code of an Ada program, without considering other information such as linker directives, it is possible to determine (within limits) its behavior as an input-output process; however, it is not possible to determine its timing. For this reason, it is necessary to use additional information, above and beyond the source code, to verify performance requirements.

OBSTACLES TO VERIFYING PERFORMANCE REQUIREMENTS

This section describes several reasons why verification of performance requirements is not a straightforward task, even given a design that has been carried to the level of compiled Ada code, and a well-defined target machine.

UNSPECIFIED MAPPING ONTO THE TARGET

Perhaps the major obstacle to verifying performance requirements on a design presented as Ada code is that lack of information concerning the mapping of the program onto the target machine. It is only in the mapping information that performance-critical issues such as the following are dealt with:

* Optimizations. These include low-level optimizations such as dead code detection and constraint check elimination, and high-level optimizations such as subprogram inlining and monitor task optimizations.

* Target resource allocation. This includes the assignment of tasks to processors (whether the assignment is static or dynamic), the allocation of data to memory (registers or main memory, resident or non-resident, and arrangement into memory banks or pages), and backup and casualty configurations.

* Implementation dependencies. These include all the implementation dependencies allowed by the Ada language definition, such as the number of task priorities, task scheduling algorithm within a priority, and interrupt handling methods.

As an example of the importance of these issues, note that it is possible to construct an Ada program that will deadlock under one legal task scheduling algorithm, but not under another legal task scheduling algorithm.

Note that a large amount of the optimization and target resource allocation data can change as a result of an apparently small change in the design. For example, the declaration of a small data object can cause the allocation by the compiler of static data to memory banks to be significantly revised, with potentially important changes to timing. This effect is particularly pronounced if a globally optimizing compiler is used.

NON-CATEGORICAL SPECIFICATIONS

A "categorical" specification is one which defines only one target system. Of course, design specifications are generally intended to be non-categorical, that is, to permit substantial freedom in their implementation.

The problem of non-categorical specifications is that if too much freedom of implementation remains, there can be a combinatorial explosion in the number of cases requiring examination in order to verify a requirement. For example, consider a target machine that consists of 3 dissimilar processors connected by communication channels. If the program contains 12 tasks, and if the design does not constrain the allocation of tasks to processors, then there are 12 to the third power (1728) configurations, each of which requires verification. Each choice left open by the design potentially multiplies the number of configurations that must be dealt with in verification.

NON-INVERTIBLE DATA DEPENDENCIES

The processing time for some operations depends on the input conditions to those operations (i.e., input data and retained data). For example, the time required by a track processing operation may depend on the number of currently live tracks. For a given operation, let the function that maps input conditions to processing time of that operation be called its data dependency function.

Data dependency functions are often invertible, at least in the rough sense that the set of input conditions that result in processing times less than some limit can be determined. For example, it might be determined that the time necessary to search a track file will be less the 25 milliseconds if there are no more the 100 live tracks to be searched. This sort of inversion of the data dependency function is often sufficient to verify whether the operation meets its performance requirements.

Unfortunately, data dependency functions are found in practice that are not invertible. That is, there are operations for which the processing time depends on the input conditions, but the dependency is too complex to invert. Phrased otherwise, it is impossible in practice to define the set of input conditions on which the operation will complete within its prescribed time.

Examples of such non-invertible data dependencies can be found in combinatorial algorithms, and in artificial intelligence paradigms. Specifically, consider a backtracking algorithm -- depth first search for an optimum value using bounding functions. It may happen that a long series of nodes will be generated and expanded before it is discovered that this series does not lead to an optimum, and must be discarded (the "garden path" phenomenon). It is not in general possible to give a simple condition defining those sets of input data that give rise to this phenomenon. In such cases, it is impossible to discriminate input conditions for which processing will be fast from input conditions for which processing will be slow.

NON-DETERMINISTIC BEHAVIOR

Non-deterministic behavior of a program is behavior that cannot be predicted from the input conditions. Non-determinism can arise from the hardware level, as when two processors race for access to a memory word, from the run-time software level, as when the operating system takes varying times to respond to a service request due to the varying activity of peripherals or to the varying activity of other programs under its purview, and from the software level, as from the Ada select statement and Ada arithmetic, which are defined as (potentially) non-deterministic.

The property of being non-deterministic differs from being non-categorical in that non-determinism may be a property of the behavior of a single system, whereas only a specification can be non-categorical. The property of being non-deterministic differs from having a non-invertible data dependency function in that the data dependency function of a non-deterministic process can only be defined statistically, and that function may or may not be invertible.

One example will suffice to demonstrate the difficulties presented by non-determinism to the verification process. Consider a program that is deterministic except that the select statement is implemented non-deterministically. That is, when several rendezvous are possible, the choice of which to accept is made at random. The state space of such a program branches each time a select with two or more open accept branches is executed. Therefore the number of distinct possible program behaviors can grow rapidly with time, and it must be verified that all these behaviors meet the requirements.

ADAPTIVE BEHAVIOR

Adaptive behavior refers to the aspects of a program's behavior that change relatively slowly over time, for the purpose of improving its performance. Examples of adaptive behaviors are load balancing functions in distributed systems, and programs that learn from experience.

Adaptive behavior can be implemented in a straightforward manner, as by changing a vector of locations, and adaptive behavior can be implemented by highly sophisticated means, as in some learning programs that, in effect, modify the code that performs some of their functions.

If the set of possible behaviors of an adaptive program is reasonably small, then adaptation causes no great problems for verification: each of the possible behaviors must be verified to satisfy the requirements. If, on the other hand, the set of possible behaviors is large, then verification may become difficult or impossible.

METHODS FOR VERIFYING PERFORMANCE REQUIREMENTS

Substantial work has been done in the area of dealing with performance requirements. SREM [3] is a method of expressing requirements, including performance requirements. SREM also provides a means to simulate the behavior of the specified system. Unfortunately for our present purposes, the SREM methodology is not well suited to producing Ada programs.

The Model system [4] generates programs (in PL/1) of a restricted form from a specification expressed in an ad hoc language. The system then estimates the performance of the resulting system, using data generated as a by-product of the program generation process together with inputs from the user on the times of the target machine for "input, output, arithmetic, comparison, and function operations."

Several methods support performance estimation based on queueing theory. Examples are PAISLey [5] and SARA [6], and Petri net approaches [7]. Such methods are effective when a network of queues is an acceptable model of the execution behavior of the software, and when statistical estimates of timing (as opposed to guaranteed worst-case values) are acceptable.

Note that none of the preceding techniques is intended to solve exactly the problem addressed by this paper: validating performance requirements on a detailed design expressed as Ada code.

One popular non-method for dealing with performance requirements needs to be noted. There is some feeling that any concern for performance is improper, almost immoral, during program design. This attitude will be called the DEMO methodology (for DEliver Me from Optimizations). The DEMO methodology calls for programs to be designed exclusively for correctness, modifiability, and maintainability, and that efficiency will taken care of later. The claim is that whatever degree of efficiency is called for can be provided, automatically, after the completion of detailed design, by one of three means:

* Compiler optimizations. "Any decent implementation" of the Ada language will provided extensive, global, optimizations, resulting in a system that will be as efficient as if it had been optimized by hand.

* Recoding hot-spots into low-level code. Since most of the execution time in many programs is taken up by a small proportion of the lines of code, those blocks of code may be recoded into assembly code, and good efficiency thereby obtained at small cost.

* Hardware. If the program does not run fast enough, a faster computer should be used. It does not matter if no such computer is available today, since it will be available soon.

The DEMO attitude probably developed in response to the older, pre-software engineering attitude that what makes software good was first, being efficient, followed closely by meeting spec, and all other values, such as maintainability, were of insufficient importance to deserve mention. If DEMO is a reaction to that attitude, it is largely justified, but nevertheless it is an overreaction. Consider each of the preceding three points:

While extensive, global, optimizations are within the state of the art, no Ada compiler known to this author provides the facilities previously demanded of "decent implementations" of the language. This is due to two factors: the demand for reasonably fast compilation, and the separate compilation facilities of the language. The result is that locally, generated code is not as particularly good, and global optimizations are not performed at all. Hence we cannot depend on compilers to solve our efficiency problems today.

Recoding of hot-spots into low-level code is of course a valuable technique as far as it goes. It does not help in two important cases: distributed inefficiency, and hot assembly code. The former refers to inefficiencies that are widely spread throughout a program; for example, a currently popular Ada compiler emits respectable code to reference arrays that have an index subtype such as 1..10, and highly inefficient code for arrays having the index subtype 0..9; no localized recoding will help. Hot assembly code refers to the case in which the program's hot-spots are in subroutines that are already in assembly code; in particular, when the hot-spots are in the run-time support code. For example, a program that is bound by task suspension and dispatch times cannot be helped by recoding into low-level code.

The hardware solution depends on cost-effectiveness. There is a balance between the cost of optimizing software, and maintaining that optimized software, against the cost of using of a faster computers, taking into account weight, power, and logistics issues. That balance cannot be casually tipped in either direction, no matter how convenient it would be for the software validation process.

The remainder of this section concerns a proposed approach to solving the problem defined above.

The basis of this approach is a change in viewpoint of the meaning of a design. A design is conventionally considered to define, roughly, an abstract computation (i.e., a function mapping inputs into outputs) together with a structure for the software. Note that the meaning of "structure of the software" is not entirely evident: the structure of the source code -- its hierarchical decomposition of a program into packages, tasks and subprograms and the separate compilation structure -- may be quite different from the structure of the software at run time. For example, code of one subprogram may be consolidated into the code of many others by means of inlining, and the program's static data may be divided up arbitrarily across several computers, and further into resident and non-resident segments. Conventionally, a "design" may specify any or all of these software structures.

For the purposes of verifying performance requirements, let us adopt the following viewpoint on the meaning of "design":

A DESIGN IS A CONSTRAINT ON THE INITIAL STATE OF THE TARGET MACHINE

That is, of the very large number of possible initial states for the target machine, a design selects a subset of those states, all of which presumably define programs that will perform according to the program's requirements. The word design will be used only in this sense below.

A design may be expressed as Ada code with annotations, or as Ada code with a separate data structure that constrains the mapping of the program onto the target machine. Examples of data that may reasonably be included in a design include the type and configuration of the target computer processors, memories, and communication channels, the mapping of static data onto memories, the mapping of tasks (or task types) to processors, and the identification of the run-time support code and parameters (such as task scheduling algorithm).

Even after the human designers have expressed all the information they have concerning the mapping of the program onto the target machine, additional information is required from the compiler concerning its mapping decisions. A form in which this information could be expressed will be presented shortly. This information includes data on the compiler's choices of optimizations, such as upmerging, inlining, and code motion.

When all of the available information on the source code and its mapping onto the target machine is available, then the verification of performance requirements can proceed. The essence of verifying performance requirements is to prove certain statements about the program behavior correct. The statements to be proven correct are the requirements ("The interval between updates to each track shall be on the average at most two seconds, and in no case longer than five seconds"), and the hypotheses are the available rules about the program and its mapping onto the target machine, together with some rules defining the behavior of the target machine itself.

Since the verification of a requirement is likely to be a long, but not particularly subtle, chain of reasoning, such verifications are likely candidates for automation. For this to be feasible, the data on the program will have to be expressed in a form acceptable to a theorem-proving system, such as a Prolog implementation [8] or a rule-based system [9]. For example, part of one set of rules presented to the verifier, which expresses the run-time structure of a subroutine, might have a semantic content (but not a form) such as

1) Subroutine S117 is completed when Block249 is completed and Loop98 is completed.

2) Loop98 is completed when Boolean4276 is false.

3) Block249 requires 79 milliseconds to complete.

4) Each iteration of Loop98 requires 182 milliseconds.

It is to be expected that attempts to verify requirements by this method will initially fail, simply because the conclusion is not justified by the available information. That is, requirements will not be validated because there is not sufficient data to establish that those requirements will be satisfied by the final system. This is as it ought to be.

When requirements cannot be validated due to the lack of sufficient data, additional information must be made available. Examples of such information would be a conclusion that is justified by the available information but is too deep for the verifier to discover (such as that some iterative process must converge within a fixed number of iterations), or information that is added to the design in order to meet the performance requirement (such as that when Condition equals Red, then the availability of Processor Alpha to Program Zeta will be 100%.) If such additional information does not permit the truth of the requirement to be deduced, then that requirement must be reported as not satisfied.

This rule-based verification approach has the following strengths:

* Accuracy. If a requirement is verified by rule-based verification, it is highly probable that any system produced according to the design will satisfy the requirement. Also, if a requirement is not verified by this method, it is highly probable that some system can be produced according to the design that will not satisfy the requirement. The method is well suited to handling worst-case requirements.

* Ability to handle non-determinism. In contrast to simulation-based approaches, the rule-based verification approach does not require that state transitions be uniquely defined: a rule stating that under certain conditions, either Process Alpha or Process Beta will be dispatched is perfectly acceptable.

* Ability to accept non-categorical specifications. A rule-based verification process is well suited to handle non-categorical specifications.

* Ability to repeat a validation following a modification. After a change to a design, such as specifying pragma inline for a function, validation may be repeated for only the cost of computer time.

This rule-based verification approach has the following weaknesses:

* Required tool support. The major tool support required to use rule-based verification is the rule-based system processor, and the additional function required of the Ada compiler (viz. emission of information on mapping decisions). Rule-based system processors are commercially available, but the modification to the Ada compiler is not trivial.

* Required human effort. Substantially more effort than is traditionally expended will be required on the part of the verifiers and the designers to achieve verification under this approach.

* Inability to handle non-invertible data dependencies. The use of a rule-based system will not solve the problem of unpredictable processing time.

* Inability to handle adaptive behavior. The use of a rule-based system will not solve the problem of unpredictable processing.

SUMMARY

Today, it is impossible to verify performance requirements on Ada software, except in a very approximate sense. There are several reasons for this difficulty, of which the main reason is the lack of use of information on the mapping of the program onto the target machine.

An approach to a partial solution to the verification of performance requirements on Ada software is here proposed, called the rule-based verification approach. This approach is suitable when the target machine is well-defined and when additional effort and expense are justified in order to guarantee that the performance requirements will be met by the final system.

REFERENCES

[1] B. W. Boehm, "Verifying and Validating Software Requirements and Design Specifications," IEEE Software, pp. 75-88, Jan. 1984.

[2] P. Zave,"The operational versus the conventional approach to software development," Communications of the ACM, pp. 104-118, Feb. 1984.

[3] M. W. Alford, "A requirements engineering methodology for real-time processing requirements," IEEE Transactions on Software Engineering, vol. SE-3, pp. 60-69, Jan. 1977.

[4] J. S. Tseng et al., "Real-Time Software Life Cycle with the Model System," IEEE Transactions on Software Engineering, vol. SE-12, pp. 358-373, Feb. 1986.

[5] P. Zave, "An operational approach to requirements specification for embedded systems," IEEE Transactions on Software Engineering, vol. SE-8, pp. 250-269, May 1982.

[6] G. Estrin et al., "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," IEEE Transactions on Software Engineering, vol. SE-12, pp. 293-311, Feb. 1986.

[7] M. K. Molloy, "Discrete time stochastic Petri nets," IEEE Transactions on Software Engineering, vol SE-11, pp. 417-423, Apr. 1985.

[8] M. R. Genesereth and M. L. Ginsberg, "Logic Programming," Communications of the ACM, vol. 28, pp. 933-941, Sept. 1985.

[9] F. Hayes-Roth, "Rule-Based Systems," Communications of the ACM, vol. 28, pp. 921-932, Sept. 1985.

N89-16359

The "Computerization" of Programming: Ada(tm)-Lessons Learned

Dennis D. Struble
Intermetrics, Inc.
733 Concord Ave.
Cambridge, MA 02138

1.0 Introduction

During the past four years, Intermetrics has constructed one of the largest systems yet written in Ada. This system is the Intermetrics Ada compiler. As you might imagine, Intermetrics has learned many lessons during the implementation of its Ada compiler. This paper describes some of these lessons, concentrating on those lessons relevant to large system implementations.

As I considered what lessons to discuss an amusing thought occured to me. Four years ago I gave a briefing at the Johnson Spacecraft Center entitled "Ada: A Management Overview." At that time, I was an ardent Ada proselytizer but one who had never laid hands on an Ada compiler. In that briefing four years ago I made several predictions about what it would be like to manage an Ada project. Having spent the last two years managing an Ada implementation, I thought I ought to determine how accurate my predictions had been. (As you might guess, my predictions turned out to be correct. If they hadn't, there certainly would have been no point in admitting to them in this paper.)

Before I identify these predictions, I'll first describe the characteristics of the Ada compiler implementation project at Intermetrics. Then after listing the predictions I will describe some specific experiences which verify these predictions.

2.0 Project Description

The Intermetrics Ada compiler and linker comprise 400,000 lines of Ada code. The compiler is augmented by a program library manager and by a set of tools which are together another 100,000 lines of Ada. The tool set includes a source lister which optionally includes the generated assembly code, a completeness checker, a body generator, the Byron(tm) design language processor, a debugger, and a set of static and dynamic program analyzers.

Ada(tm) is a registered trademark of the U.S. Government (Ada Joint Program Office).
Byron(tm) is a trademark of Intermetrics, Inc.

Intermetrics is currently completing a total of six compilers under two government contracts and four commercial contracts. The compilers generate code for the IBM 370, the Sperry 1100, and the MIL-STD-1750A instruction sets; this generated code executes in six different run-time environments: IBM MVS(tm), IBM CMS(tm), Amdahl UTS(tm), Sperry 1100, and bare 1750A. The compilers are hosted under four different operating systems: IBM MVS, IBM CMS, Amdahl UTS, Sperry 1100, and VAX VMS(tm).

All of these compilers have been developed in parallel and all of the compilers share the same source code. The source code is maintained under a configuration management system designed specifically to support a multi-hosted and multi-targeted compiler development environment. The development staff at its peak included fifty software engineers.

The development environment for the Ada compilers is an IBM 3083 Model BX, running the Amdahl UTS operating system hosted under VM. The production-quality compiler was initially developed using an Ada-subset compiler Intermetrics wrote in Pascal. The production-quality compiler was validated in December 1985 and was bootstrapped through itself in February 1986.

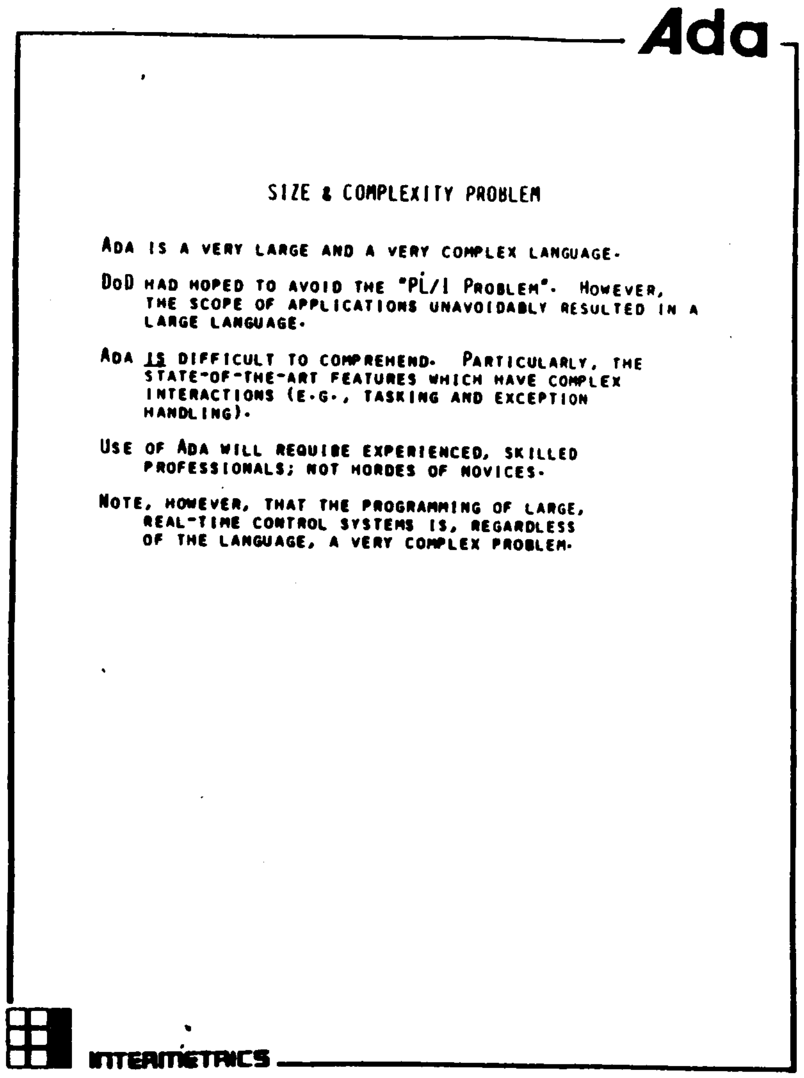
3.0 The Predictions

Figures 3-1 and 3-2 are extracted from the four-year old briefing I described above. The predictions contained in these figures are self-explanatory. Of these predictions, the ones concerning multi-tasking are, of course, not relevant to our compiler. (Not yet at least; Intermetrics is anxious to modify our compiler to become the first Ada compiler to take advantage of the new generation of multi-micro machines.)

All the other predictions have turned out to be more or less correct. One theme that runs through these predictions is that with the introduction of Ada, the DoD is attempting to take a major step forward in the "computerization of programming." I use the term computerization of programming, rather than "automatic programming" because I believe that for completely new applications, such as Ada compilers and Space Station software, automatic programming will never occur. On the other hand, many of the tasks required in the programming of new systems are amenable to much greater computerization. In particular, Ada requires much more "bookkeeping" to be performed by the compiler than do other languages.

IBM(tm), MVS(tm) and CMS(tm) are trademarks of the International Business Machines Corporation.
UTS(tm) is a trademark of the Amdahl Corporation.
VAX(tm) and VMS(tm) are trademarks of the Digital Equipment Corporation.

ORIGINAL PAGE IS
OF POOR QUALITY



Ada

SIZE & COMPLEXITY PROBLEM

ADA IS A VERY LARGE AND A VERY COMPLEX LANGUAGE.

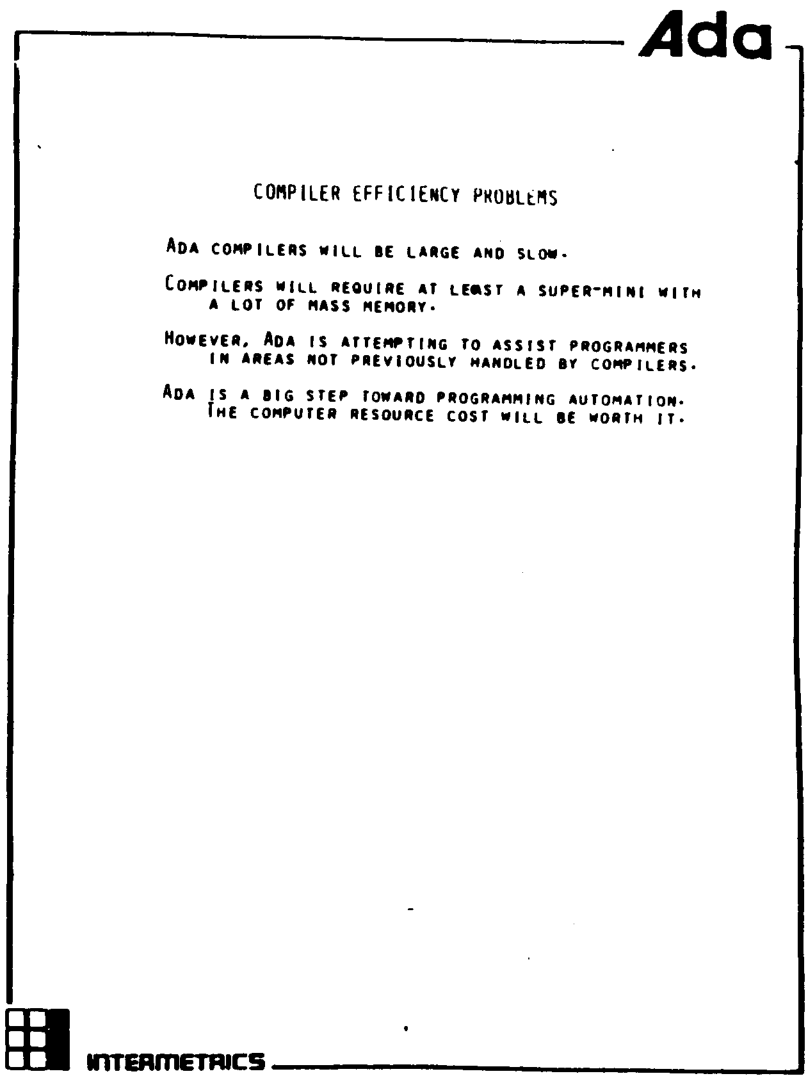
DOD HAD HOPED TO AVOID THE "PL/I PROBLEM". HOWEVER, THE SCOPE OF APPLICATIONS UNAVOIDABLY RESULTED IN A LARGE LANGUAGE.

ADA IS DIFFICULT TO COMPREHEND. PARTICULARLY, THE STATE-OF-THE-ART FEATURES WHICH HAVE COMPLEX INTERACTIONS (E.G., TASKING AND EXCEPTION HANDLING).

USE OF ADA WILL REQUIRE EXPERIENCED, SKILLED PROFESSIONALS; NOT HORDES OF NOVICES.

NOTE, HOWEVER, THAT THE PROGRAMMING OF LARGE, REAL-TIME CONTROL SYSTEMS IS, REGARDLESS OF THE LANGUAGE, A VERY COMPLEX PROBLEM.

INTERMETRICS



Ada

COMPILER EFFICIENCY PROBLEMS

ADA COMPILERS WILL BE LARGE AND SLOW.

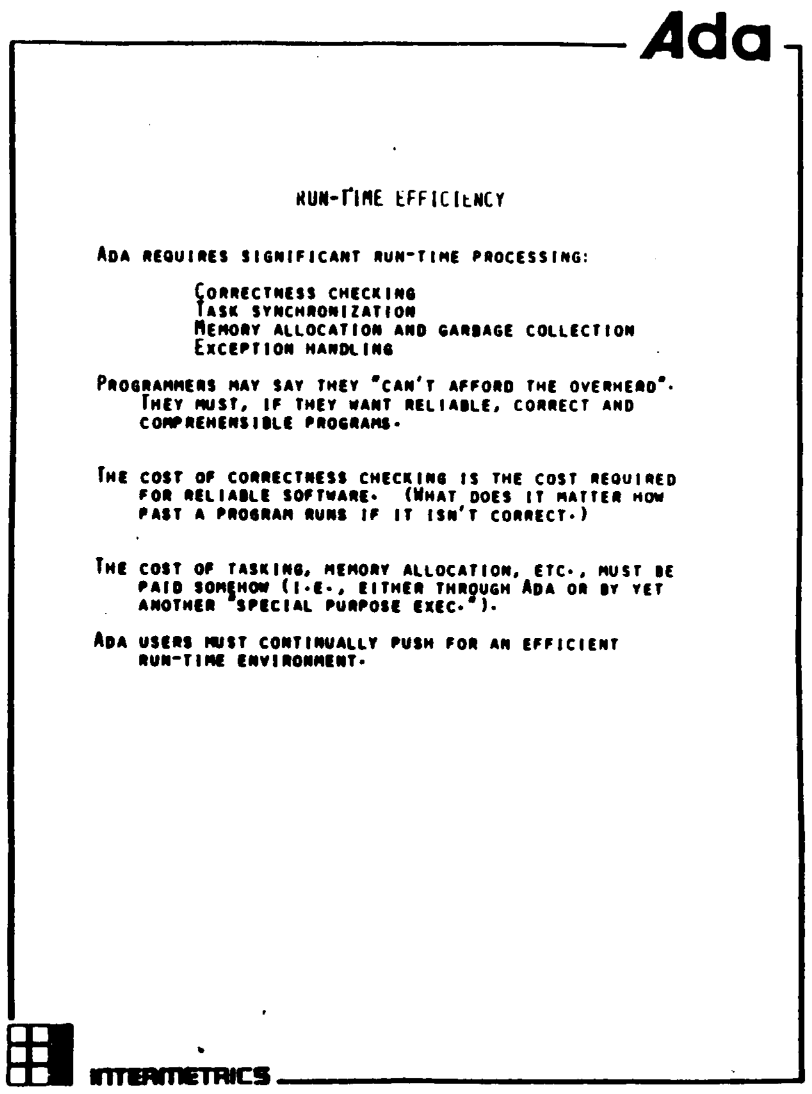
COMPILERS WILL REQUIRE AT LEAST A SUPER-MINI WITH A LOT OF MASS MEMORY.

HOWEVER, ADA IS ATTEMPTING TO ASSIST PROGRAMMERS IN AREAS NOT PREVIOUSLY HANDLED BY COMPILERS.

ADA IS A BIG STEP TOWARD PROGRAMMING AUTOMATION. THE COMPUTER RESOURCE COST WILL BE WORTH IT.

INTERMETRICS

FIGURE 3-1: 1982 PREDICTIONS (1 of 2)



Ada

RUN-TIME EFFICIENCY

ADA REQUIRES SIGNIFICANT RUN-TIME PROCESSING:

- CORRECTNESS CHECKING
- TASK SYNCHRONIZATION
- MEMORY ALLOCATION AND GARBAGE COLLECTION
- EXCEPTION HANDLING

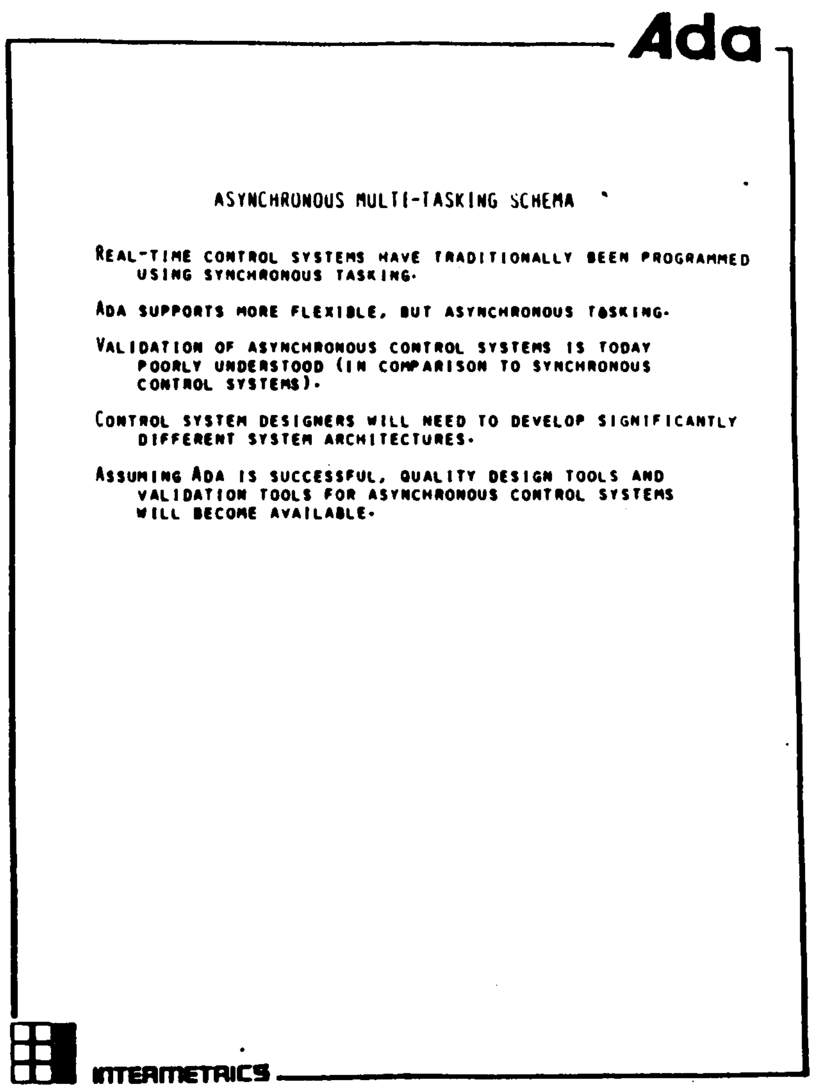
PROGRAMMERS MAY SAY THEY "CAN'T AFFORD THE OVERHEAD". THEY MUST, IF THEY WANT RELIABLE, CORRECT AND COMPREHENSIBLE PROGRAMS.

THE COST OF CORRECTNESS CHECKING IS THE COST REQUIRED FOR RELIABLE SOFTWARE. (WHAT DOES IT MATTER HOW FAST A PROGRAM RUNS IF IT ISN'T CORRECT.)

THE COST OF TASKING, MEMORY ALLOCATION, ETC., MUST BE PAID SOMEHOW (I.E., EITHER THROUGH ADA OR BY YET ANOTHER "SPECIAL PURPOSE EXEC.").

ADA USERS MUST CONTINUALLY PUSH FOR AN EFFICIENT RUN-TIME ENVIRONMENT.

INTERMETRICS



Ada

ASYNCHRONOUS MULTI-TASKING SCHEMA

REAL-TIME CONTROL SYSTEMS HAVE TRADITIONALLY BEEN PROGRAMMED USING SYNCHRONOUS TASKING.

ADA SUPPORTS MORE FLEXIBLE, BUT ASYNCHRONOUS TASKING.

VALIDATION OF ASYNCHRONOUS CONTROL SYSTEMS IS TODAY POORLY UNDERSTOOD (IN COMPARISON TO SYNCHRONOUS CONTROL SYSTEMS).

CONTROL SYSTEM DESIGNERS WILL NEED TO DEVELOP SIGNIFICANTLY DIFFERENT SYSTEM ARCHITECTURES.

ASSUMING ADA IS SUCCESSFUL, QUALITY DESIGN TOOLS AND VALIDATION TOOLS FOR ASYNCHRONOUS CONTROL SYSTEMS WILL BECOME AVAILABLE.

INTERMETRICS

FIGURE 3-2: 1982 PREDICTIONS (2 of 2)

A more significant computerization of programming arises because Ada fosters, if not requires, a database management approach to the handling of software. That is, each Ada package should be treated as a valuable, complex, and evolving piece of data; database management facilities and procedures should be provided that are commensurate with the value and complexity of this data.

As Intermetrics has further computerized its software implementation procedures through the use of Ada, Intermetrics has learned several lessons which confirm those four-year-old predictions, as well as some lessons that could not have been anticipated four years ago. These lessons are described below.

4.0 Ada-Lessons Learned

The lessons Intermetrics has learned may be split into the following categories: Ada Training, Ada Tools, and Ada Language Use.

4.1 Ada Training

One of the predictions states that the use of Ada would required well-educated software engineers. Implied by this prediction is a possible short-fall in software engineers trained in Ada and trained in the software engineering principles that Ada encourages.

In fact, availability of trained Ada engineers has not been a problem at Intermetrics. This is because the Intermetrics Software Systems Group employees computer scientists who specialize in support software. Most of our new employees already know Ada and already know the system design principals associated with Ada software engineering.

Ironically, in some cases this broad knowledge of modern language technology has actually caused problems. Some engineers who have worked with university-developed, state-of-the-art languages expect Ada to behave the same way. Many of these state-of-the-art languages emphasize expressability, perhaps at the expense of run-time efficiency, whereas run-time efficiency was a key criteria in the design of Ada (and has been a key criteria in the development of the Intermetrics Ada compilers.)

An example of the problems caused by an orientation to state-of-the-art languages arises from the CLU programming style which advocates regular use of "signals" to return status from subprograms. Several new Intermetrics employees have assumed that in a corresponding way, exceptions should be used in Ada programming to return subprogram completion status. In fact, Ada exceptions are intended for truly "exceptional" circumstances. Efficient Ada compilers attempt to generate code in such a way that exceptions require no processing time unless, and until, the exception is signalled. However, when the exception is signalled, substantially more processing is required than simply returning an output parameter. Thus, use of Ada exceptions is not analogous to use of CLU signals. Through coding standards and code reviews, Intermetrics educates its programmers into efficient use Ada programming.

4.2 Tool-Use Lessons

In using high-order languages, Intermetrics often has found that the quality of the compiler is more important than the quality of the language. Certainly in the initial years of Ada use, this will be the case. Three characteristics of Ada tool usage are discussed below: the importance of the library manager, the unfortunate variability among Ada compilers, and the substantial computing resources required by Ada tools.

4.2.1 A Sophisticated Library Manager is Critical

During the parallel construction of the six compilers, all of which share the same Ada program library, the necessity for a database management approach to Ada software configuration management became clear. It is the Ada program library manager that provides this database management. This database manager must provide the following services:

Separate development areas for projects and sub-projects along with a facility to share formally "released" packages among projects and sub-projects.

Management of variants of subsystems, where these variants support rehosting or retargeting the overall system.

Formal configuration management of successive versions of subsystems.

An interactive facility that can answer queries concerning the status of packages in the library as well as queries concerning dependencies among packages.

An interactive facility which supports constructing a system by choosing specific variants and versions for each sub-system.

4.2.2 All Ada's are <u>not</u> the same

During the developement of its Ada compilers, Intermetrics has used three different Ada compilers and attempted to use a fourth. The three successfully used compilers are the two Intermetrics compilers and the DEC (tm) compiler. (One Intermetrics compiler and the DEC compiler are validated compilers.) Not surprisingly, these compilers do exhibit enough variation that rehosting a large system from one compiler to another is a substantial undertaking. Some of the major differences Intermetrics encountered are listed below.

Three classes of differences were experienced: functional, capcity, and performance. Two functional differences were noteworthy; the first arises because Ada does not specify a default elaboration order. Thus, unless pragma elaborate is used exhaustively to explicitly order the complete elaboration, a complex system may elaborate correctly using one compiler and yet fail to elaborate using another.

The more troublesome functional problem involved the different handling of un-initialized records. It is, of course, incorrect to rely on un-initialized variables. Nevertheless, it is common in large systems developed using a compiler that <u>does</u> initialize all variables to zero by default, that this large system will work correctly even though some variables are not explicitly initialized. When such a large system is rehosted to a compiler with a different default initialization, it becomes extremely costly to identify the un-initialized objects.

At times potential customers have asked us to rehost our compiler front-end and Byron tool set to systems already having an Ada compiler. In one case we were unable to respond to the request because the existing compiler did not have the capacity necessary to compile the largest units in the Intermetrics compiler. (Generally, the Intermetrics compilation units are from ten to several hundred lines; however, there are a few very large packages in the compiler. These packages include the parser tables, the code-generator tables, and the DIANA access package.)

The most serious difference we encountered was the speed of our compiler as compiled by different compilers. We, of course, expected variation in the code quality among the different compilers; when we forecast the speed of our compiler on the VAX as compiled by the DEC Ada compiler, we took into account the difference in code quality and difference in machine speed. Nevertheless, our I/O-intensive, host-interface package, which conforms to the CAIS file model, ran much more slowly on the VAX than anticipated. We eventually identified Ada file open and close operations as the cause of this anomaly. The lesson is that for extrapolating the performance of a systems-level Ada program, a simple comparison of code-quality is not sufficient.

There are straightforward procedures which may be used to avoid these compiler variability problems. Foremost is the identification of those aspects of Ada which may vary from compiler to compiler and establishment of coding standards addressing these variations. If you know in advance that your system will be rehosted to several compilers, investment in a standards checker will definitely pay off.

For a large project such as the Space Station which will have the resources to modify its compilers, it would be appropriate to enhance each compiler to flag possible sources of incompatibilities and to generate code that conforms with the anomalies of other compilers. For example, Intermetrics is considering adding a DEC-Ada compatibility option to the Intermetrics compiler so that we may minimize the recurring cost of rehosting the Intermetrics compiler to the VAX.

4.2.3 For Ada, Don't Underestimate the Computes!

Sure enough, Ada compilers have turned out to be big and to be slow. Despite what some may hope, an Ada compiler will always be slower than an equivalent Pascal or C compiler; it's a simple issue of algorithmic complexity. Again, Ada is attempting to computerize software engineering substantially more than have previous languages; this computerization requires substantial computing resources.

4.2.3.1 Compile and Link Speed

All potential Ada users are aware that average compilation speed is a critical compiler characteristic. Nevertheless, in addition to the average lines-per-minute speed of Ada compilers there are several other compilation speed issues that are unique to Ada. These are start-up overhead, speed of separate compilation, and up-to-dateness checking.

Ada compilers have a start-up overhead greater than previous compilers. This arises from the size of the compiler executable and from the requirement to interact with a large database, namely, the program library. Consequently, the cost of compiling very small modules is greater than with previous compilers. This cost should be taken into account when estimating computing resource requirements and perhaps when partitioning your system into compilation units.

One Ada's most valueable characteristics is its requirement that the compiler verify module interfaces. Once again, this further computerization requires processing time. Each package that a given package "with's" must be accessed and its interface information made available to the current compilation. Extended chains of "with" dependencies across packages add further accessing cost. Thus, the hierarchical structure of large systems must be designed carefully to avoid including extraneous dependencies among packages. Further the dependency structure should be periodically re-assessed during a long implementation effort to determine if adjustments to this structure would improve compilation time.

Ada compilers and linkers are required to check the "up-to-dateness" of Ada packages. In a large system with a complex library structure, the look-up required to verify up-to-dateness will be significant. Again an understanding of this issue is important when evaluating Ada compilers and when estimating required computer resources.

4.2.3.2 Disk Storage Requirements

Systems written in Ada will require substantially more disk storage than previous systems. This arises from two factors. First Ada requires a program library that maintains interface information from preceding compilations. Secondly, and more importantly, some Ada compilers, including the Intermetrics Ada compilers, provide an open interface into the internal data structures that describe the packages of the compiled system. The Intermetrics Ada compilers provide this open interface through DIANA. A program library containing a DIANA description of each package in the system enables the construction of a set of tools that can analyze these packages. These tools include static analyzers, dynamic analyzers, debuggers, package status reporting tools, and package documentation tools. An advantage of an open interface is that a given project, like the Space Station, can readily implement whatever analysis tools the project requires.

This open interface facility does have a computer resource cost, namely more disk storage than required by previous languages. In evaluating this cost, managers must recall that with the advent of Ada compilers which provide a DIANA-based program library, we are taking a significant step toward a database-oriented view of software systems. Such a methodology does imply the disk storage resources required for a large database.

Recognizing that a given project may not want to provide the resources necessary for a complete DIANA database, the Intermetrics Ada compilers will provide the option to retain only enough DIANA to support Ada interface checking. Even though Intermetrics will provide this option, we do anticipate that most projects will find the benefit provided by the DIANA-based toolset will substantially outweigh the cost of the disk storage.

It is interesting to note that the issue of program library size and program library functionality is only slowing beginning to appear in various Ada compiler evaluation criteria. This is because a sophisticated program librarian and its disk storage requirements were never an issue with Ada's predecessors. With Ada, the characteristics of the program library may well become one of the key distinguishing characteristics of Ada compilers. The functionality of the library will determine how effectively a large number of programmers will be supported and how effectively parallel development efforts will be supported. The size of the program library will be an important parameter when a manager budgets for computer resources.

4.3 Language-Usage Lessons

Building one of the first large systems in Ada is like attending a grand buffet banquet in a foreign country. There's a table full of goodies that look incredibly delicious. The problem is: some of the goodies may not agree with you and there are so many goodies it would be very easy to overeat. Listed below are some of the Ada features that in some case turned out to be a little too rich.

4.3.1 Beware Abstraction Overdose!

From its inception, the Intermetrics compiler was designed and coded fully utilizing Ada's excellent support for data abstraction. Each of the compiler's major data structures is designed as a data abstraction with an appropriate set of access procedures. The compiler's heavy reliance on abstractions has worked out well from both the robustness and flexibility standpoints.

For example, the compiler was designed with a software paging system that would manage the storage for the various intermediate languages. During the first year, while the paging system was being the implemented, a simple, memory-resident system was used as a substitute. When the time came to switch over to the paging system, we anticipated a lengthy integration and debugging phase. However, because the underlying implementation of the storage primitives had been hidden, the switch-over phase proceeded with almost no bugs.

Data abstraction does, however, have a negative side: data abstraction, particularly if overused, can substantially degrade a system's performance. Going through multiple levels of abstraction, each one of which is a procedure call, is expensive. As we complete our compiler, we find ourselves having to "collapse" some of these levels, specifically, the parser's access to the parse tables and the code generator's access to the code tables.

Having experienced both the benefits and costs of heavy use of data abstraction, we believe the best approach is to start out with those abstractions that best support initial development and integration. However, a project manager must definitely budget time and effort to measure the cost of abstraction usage once the system has been integrated. And unfortunately, a project staff probably will need to tune some of the abstraction usage in order to meet the project's performance requirements.

4.3.2 Don't Touch that Spec (and leave my body alone too!)

A key Ada design prinicipal is the physical separation of package specification code from package implementation code. An intended benefit of this separation is the avoidance of re-compilation that could result from changes to the implementation code. Intermetrics experience shows, however, that the simple division into spec's & bodies does not guarantee minimal compilation.

To assure minimal re-compilation, management diligence is required. Ada's strong interface checking has its downside. In C, Pascal, or FORTRAN, modules are not strongly connected and hence modules may be recompiled readily. In Ada, packages are very strongly connected and if changes to packages are not managed, one can spend enormous amounts of computer dollars re-compiling.

The strongly connected aspect of Ada necessitates a software development approach that emphasizes bottom-up coding and unit testing. The hierarchy of packages must be built in a manner that freezes the interfaces and thereby prevents undesired recompilations. This development approach is, of course, a standard aspect of good software engineering and most projects do attempt to adhere to this approach. Nevertheless, when using Ada, the cost of not following this approach become greater since Ada will force recompilations whenever the interfaces appear to have changed (even if the programmer knows they haven't).

Another aspect of interface management arises because Ada's spec and body separation is not as strong as normally believed. Changes to generic bodies and to in-lined procedures will cause recompilation. Consequently, managers must make sure that the staff is aware of these possible body dependencies and structure their packages to minimize re-compilation necessitated by changes to both spec's and bodies.

In addition to fostering a package partitioning that minimizes recompilation, a manager should also make sure the project's APSE includes a what-if analyzer. A what-if analyzer answers the question: "What recompilation would result if I make the following change to this spec or to this body." This tool is particularly valuable during maintenance when a substantial change, for example for performance reasons, is being contemplated. It is likely that a maintainer would not fully understand the recompilation dependencies in a large system. A what-if analyzer could guide the design toward one which avoids substantial recompilation.

4.3.3 Lady Lovelace, she doth nag.

Ada's pervasive constraint checking is thought by many to be a meddlesome annoyance best handled by liberal use of pragma suppress. Intermetrics did not agree with this view at the inception of its Ada development and our experience has shown our perception to be correct.

Constraint checking has been perhaps the most valuable Ada feature we've enjoyed during the compiler's development. The positive attributes of contraint checking include:

> Bugs manifest themselves very close to their "time of occurrence." In developing a compiler this is critical, since the generation of incorrect code, when undetected, produces the most difficult bugs. Fortunately, ninety percent of the time, our compiler failed with a constraint check rather than blithely generating incorrect code.
>
> By providing appropriate exeception handlers, bug occurrences can be made somewhat self-documenting. That is, an exception handler can identify the context in which the constraint error occured. For example, when a contraint error occurs in our compiler, it prints out the line number of the source line being compiled and dumps the relevant internal data structures. (This contrasts with the more conventional, unadorned "memory exception" and "operation exception".)
>
> Given self-documenting failures, contraint checking allows an independent test group to play a much more active and productive role in the checkout and debug process.

Because of the value of constraint checking, Intermetrics took special care to design an optimizer that would remove all unnecessary constraint checks. Unfortunately, with constraint checking, the compiler can't do the whole job. Minimization of constraint checking also requires good Ada programming. Precise type definition is critical to avoid unnecessary constraint checks. A carefully written Ada program compiled by a good Ada compiler should result in no more checking-code than would an equivalent C program containing that amount of assertion checking mandated by good software engineering standards.

While we were using our subset compiler for development, we were concerned with the possibly unacceptable amount of constraint checking that would exist in the completed compiler. Fortunately, we were quite pleased with the contrast between no constraint check elimination in the subset compiler and excellent constraint check elimination in the production compiler. In fact, Intermetrics currently plans to achieve its performance requirements without resorting to pragma suppress. Retaining the necessary constraint checks in the compiler will markedly improve the maintainability of the compiler.

4.3.4 Look Ma - No Regressions!

The problem of regressions is indeed lessened in Ada. Prior to Ada it was often the case that in fixing a bug in a large, complex system other bugs were introduced into the system. The strong structuring support and strong typing that Ada provides make it more difficult to introduce incorrect fixes into a large system.

This characteristic of a system written in Ada was clearly indicated during both the validation and the bootstrap of our compiler. We had expected, based on prior compiler experience, that we would experience a two or three week "tail" at the end of our pre-validation testing. This tail would occur as we attempted to pass the final five percent of the ACVC suite. We expected that a fix introduced to pass one of the last ACVC's would cause one or two previously passing ACVC's to begin to fail. In fact, this regression did not occur. Our rate of getting new ACVC's to pass remained high right up through the week in which the last ACVC's were passed.

A similar phenomenon occurred when we bootstrapped our compiler. To manage the bootstrap process, we decided that we would first bootstrap the smallest compiler phase, using this mini-bootstrap to expose the majority of compiler bugs we would experience during the full bootstrap. This smallest phase is the 70,000 line, global optimizer phase. Its bootstrap required three months. During the three months 55 bugs were exposed and fixed. This bug rate corresponds to 8 bugs for each new 10,000 lines of new code exposed to the compiler.

In forecasting the bootstrap of the remaining 330,000 lines of the compiler, we estimated that these new lines would produce bugs at 4 bugs per 10,000 lines, for a total of 130 bugs. Given this number of bugs, we estimated it would require twelve weeks to bootstrap the entire compiler. To our pleasant surprise, we bootstrapped the compiler in five weeks and the additional 330,000 lines exposed only 10 new bugs!

We attribute these two instances of fewer bugs than expected to the "correctness" discipline which arises from programming in Ada. Ada does indeed appear to result in systems which have few "lingering" bugs and are readily maintainable.

5.0 Conclusion

Intermetrics realized five years ago that writing a production quality Ada compiler would be a tough job. Writing the compiler in Ada itself made the job really tough.

This heightened difficulty arose not because Ada isn't an excellent systems programming language. The difficulty arose from a situation which occurs too often in our industry: the dependence on a brand-new programming support environment for a large systems programming effort.

Fortunately, this situation is behind us. Intermetrics has a production quality, programming support environment that efficiently supports continued development of the Intermetrics Ada compilers. Intermetrics has also learned a great deal from its 150 person-years of Ada development; hopefully, the lessons described in this paper will benefit the planning and implementation of the Space Station software.

N89-16360

A small evaluation suite for Ada compilers

Randy Wilke, Daniel Roy

1 INTRODUCTION

After completing a small Ada pilot project (OCC simulator) for the Multi Satellite Operations Control Center (MSOCC) at Goddard last year, we recommended the use of Ada to develop OCCs.

To help MSOCC transition toward Ada, we recently developed a suite of about 100 evaluation programs which can be used to assess Ada compilers, namely:

- o Compare the overall quality of the compilation system (e.g., ease of use, complexity, impact on the host computer, error message quality).
- o Compare the relative efficiencies of the compilers and the environments in which they work (e.g., how long does it take to compile and link a program?).
- o Compare the size and execution speed of generated machine code.

Another goal of the benchmark software was to provide MSOCC system developers with rough timing estimate for the purpose of predicting performance of future systems written in Ada.

2 SUITE DESCRIPTION

Two types of benchmarks were created, "static" and "dynamic". Static benchmarks are used to assess the extent to which a compiler helps (or hinders) the software development effort. Dynamic benchmarks measure the efficiency of machine code generated by Ada compilers.

The Ada evaluation suite was developed in about 4 man-months on a Digital Equipment Corporation (DEC) VAX-11/785 using the DEC Ada Compilation System (V1.0) running under the VMS operating system (V4.2). The evaluation suite source was then ported from the VAX to a Data General Corporation (DG) MV/4000 via magnetic tape. The software was rebuilt on the MV/4000 using the DG Ada Development Environment (V2.3) running under the AOS/VS (V6.3) operating system.

2.1 Static Benchmark Programs

Two general classes of static evaluation programs were generated. The first set of programs measures the time to compile various Ada constructs such as:

- o A null program to measure the minimum overhead.
- o A program instantiating INTEGER_IO.
- o A program translated from Reference 2, dealing with stride and non-stride array references.
- o The DHRYSTONE synthetic benchmark program from Reference 3.

A compilation command procedure automatically measures the compile time for every program of the benchmark suite.

The second set of static benchmark programs contain deliberately induced errors in the source code. They are used to subjectively evaluate how well compiler messages help the programmer identify some common mistakes such as:

- o Incorrect dereferencing of an object in a procedure call.
- o Confusing type and subtype declarations.
- o Common typos (missing "--" and ";", reference to a misspelled variable, etc.)
- o Forgetting to qualify items from "withed" packages. In this case, a good message should mention the right package(s).

2.2 Dynamic Benchmark Programs

The dynamic benchmark programs measure the run time overhead for the following Ada features:

- Control structures (CASE, IF-THEN-ELSE, LOOP).
- Assignment statements including ACCESS types.
- Procedure call overhead (including calling another language from Ada).
- Dynamic memory allocation.

- Sequential IO.

- Rendezvous (inter-task communication) and task activation.

- Using multi-tasking to overlap IO with CPU intensive processing.

- Array referencing (stride and non-stride).

The chosen limited set of tests concentrates on the Ada language features that are vital to MSOCC. However, the benchmark methodology and the benchmark code structure provide a good framework to easily create new benchmarks as the need arises.

An averaging technique is used to smooth the effects of random system events that can be minimized but not eliminated from the multi-programming environment. A "null" loop is timed for several iterations to compute the overhead for the loop. The ADA construct to be benchmarked is then timed inside the same loop. The null loop time is subtracted from the time of the loop containing the Ada construct, and the result is divided by the number of iterations to produce the time for one execution of the ADA construct. All timing is performed using the CALENDAR.CLOCK routine.

A command procedure automatically logs all sysgen parameters as well as the main process parameters (quotas, working set, etc.) before running all tests with a programmable number of iterations. Timing results are computed internally by every benchmark program and logged in individual files (one such file per test).

2.2.1 Parallelism Test Programs Description

The programs that test the overlapping of input, output and CPU processing with tasking warrant a more detailed discussion:

2.2.1.1 PAR_BIG

This program instantiates the SEQUENTIAL_IO package for a file of big record size (10_000 bytes per block) and reads, processes and writes several records, overlapping sequential access input, CPU intensive "processing" and sequential access output by using Ada tasking with rendezvous. The overall run time should be compared to the overall run time for SER_BIG described below.

If the compiler correctly implements the Ada tasking paradigm, the processing task should be able to run while the I/O tasks are blocked. Therefore, PAR_BIG should run faster than SER_BIG provided that the rendezvous overhead is acceptable.

2.2.1.2 SER_BIG

This program instantiates the SEQUENTIAL_IO package for a file of big record size (10_000 bytes per block) to serialize sequential access input, CPU intensive "processing" and sequential access output in a loop.

2.2.1.3 PAR_NB And SER_NB

The same principles were applied to a file of nascom blocks (600 bytes). However, because modern operating systems very efficiently buffer the data during sequential IO operations, the efficiency advantage of tasking may be small (or non existent) for this test.

2.3 Code Optimization Issues

One major concern, when doing simple dynamic benchmarks, is the compiler optimizer. Most simple benchmark programs do not do any reasonable work. One must be careful that the optimizer does not recognize this fact and optimize the construct being benchmarked completely out of the program. Even if the construct is still present, there is concern as to whether the optimization would have taken place in a "real" program to the extent that it took place in the simple benchmark (e.g., all variables used in the benchmark ending up in registers may not be realistic).

The DEC Ada Compiler has two optimization switches. One, /OPTIMIZE=TIME will automatically treat small subroutines as though the INLINE pragma had been invoked. The other, /OPTIMIZE=SIZE performs all other optimization but does not do automatic INLINE processing. The /OPTIMIZE=TIME switch does not result in automatic INLINE processing if the body of the subroutine being called is compiled separately.

We tried a method described in Reference 4 to trick the compiler into not performing automatic INLINE processing. We rejected the method because it introduced large delays that would have made timing measurements of small constructs very imprecise.

We compiled all dynamic benchmarks with and without optimization. Where significant differences resulted, the generated machine code was examined to determine if the optimizer did its job "too well". In such cases, the non-optimized version was used in test runs.

3 COMPARING DEC ACS AND DG ADE

We were guests on both of the host machines and hence, were assigned limited resources. Consequently, much effort was spent managing resources, particularly disk space. On the ADE we were frequently running at reduced priority, relative to all other system users.

This comparison between the ACS and the ADE is, perhaps, a little unfair to the ADE. The VAX-11/785, which the ACS runs on, is about twice as fast as the MV/4000 (1.2 MIPS vs 0.6 MIPS). Also, while DG's AOS/VS is far superior to many operating systems, we believe that DEC's VMS, in general, provides a significantly better software development environment. These bias must be taken into consideration.

All static and dynamic benchmarks were developed on the DEC VAX-11/785 and ported to the DG MV/4000. There were no cases where the ADE failed to compile a program that was successfully compiled under the ACS. There was one instance where the ADE generated incorrect code, and one program experienced runtime problems that were never solved. Specifically, the following problems were encountered while porting the benchmark suite:

- Due to bad code being generated for an explicit type conversion, PAR_NB had to be recoded.

- PAR_BIG never ran successfully on the MV/4000.

- File IO and parallel processing programs had to be modified on the the MV/4000 because the ADE does not handle representation clauses for type "byte" and generated code for 32 bit integer instead.

- An unhandled exception would randomly occur while using a program (written in Ada) to unpack records from files that had been transferred to the DG. The problem would go away by rerunning the program with exactly the same input file.

The following additional subjective comparisons can be made:

1. Both systems use a lot of resources. The ADE makes extravagant use of disk space and is also a CPU hog.

2. The MV/4000 text editor (SED) didn't seem as friendly as the VAX's (EDT). This may have been due to lack of DG experience on the part of the evaluators (we did not know how to use SLATE).

As a rule, setting up command files to build and run things frequently took an order of magnitude longer on the MV/4000.

3.1 Static Evaluation

3.1.1 Compilation Times

ACS and ADE compilation times for a subset of the benchmark suite are compared in Figure F.3.3-1. For the sample, the ACS performed better even if we allow for the difference in processor speeds. Differences in the time required to perform disk IO is an additional, hard to quantify factor.

The entire benchmark suite was compiled and linked in less than 40 minutes on the ACS and in about 3 hours on the ADE.

	COMPILE TIME (seconds)	
Benchmark	ACS (VAX 11-785)	ADE (MV/4000)
COMP_NULL	6	24
COMP_COMMENTS	5	32
COMP_INT_IO	11	65
COMP_TEXT_IO	7	20
ARRAY_REF	31	142
MODULO_BYTE	28	92
RV_ARRAY_100	20	94
SUB_CALL_0	18	105
PAR_BIG	69	264

Figure F.3.3-1, A Sample of Compilation Times.

3.1.2 Error Messages

Even though the ACS compile time messages were verbose at times, their relevance and clarity were judged superior to those of the ADE.

In particular, the ACS makes generally good suggestions (adding missing semicolons, guessing package name for missing qualification, etc.) whereas the ADE suggested that a derived type was intended when the problem was a confusion between type and subtype declarations. This kind of suggestion can greatly confuse the novice programmer.

3.2 Dynamic Evaluation

Overall, the DEC ACS produced more efficient code than the DG ADE. The rest of this section compares execution speeds for several classes of benchmarks.

3.2.1 Common Features

Figure F.3.3-2 shows the measured run time for the most common Ada constructs.

CONSTRUCT	ACS/ADE OVERHEAD (microsec)		
	average	low	high
Control			
3 CASES	2.6/1.5	2.6/0.8	2.6/1.9
10 CASES	2.9/1.3	2.9/1.3	2.9/1.6
IF/THEN/ELSE	4.6/1.6	4.4/1.6	4.7/1.6
FOR LOOP (optimized)	1.5/6.0	1.5/6.0	1.7/6.0
Assignments			
VARIABLE := VARIABLE	0.7/3.4	0.6/3.4	0.7/3.5
ACCESS VARIABLE := VARIABLE	3.0/5.4	3.0/5.0	3.2/5.4
VARIABLE := CONSTANT (CONST < 2**8)	0.7/2.6	0.7/2.6	0.8/2.6
VARIABLE := CONSTANT 2**8 < CONST < 2**16	1.1/2.6	1.1/2.5	1.3/2.6
VARIABLE := CONSTANT (CONST > 2**16)	1.0/2.9	0.9/2.9	1.1/2.9
Synthetic benchmark			
DHRYSTONE	1.3/4.6	1.1/4.6	1.7/4.6

Figure F.3.3-2, Common Ada construct run time overhead.

3.2.2 Procedure Call

Figure F.3.3-3 shows the run time overhead for procedure calls.

NUMBER OF PARAMETERS	PARAMETER TYPE	ACS/ADE CALL OVERHEAD (microsec/call)		
		average	low	high
0	-	13/31	13/31	13/31
1	IN	17/37	16/36	17/37
1	OUT	16/37	16/37	16/37
1	INOUT	20/40	19/40	20/40
1 (C calls C)	IN	13/NA	13/NA	14/NA
1 (Ada calls C)	IN	15/NA	15/NA	16/NA
10	IN	56/89	56/89	56/172
10	OUT	55/89	55/89	55/90
10	INOUT	86/121	86/121	86/124
10 element array	IN	14/33	14/33	14/34
10 element array	OUT	14/34	14/33	14/35
10 element array	INOUT	14/34	14/33	14/35
100 element array	IN	14/33	14/33	14/33
1000 element array	IN	14/34	14/34	14/34
10000 element array	IN	14/NA	14/NA	14/NA

Figure F.3.3-3, Procedure Call Overhead.

3.2.3 Memory Allocation

Figure F.3.3-4 shows the overhead measured for dynamic memory allocation.

NUMBER OF BUFFERS	SIZE OF BUFFERS	ACS/ADE ALLOCATION OVERHEAD (millisec/allocation)		
	(bytes)	average	low	high
100	1000	0.9/5.0	0.8/4.0	1.2/5.0
500	1000	2.9/4.6	3.6/4.6	2.8/4.6
1000	100	0.2/1.5	0.2/1.5	0.2/1.5
1000	500	6.4/4.7	6.5/4.6	6.2/4.9

Figure F.3.3-4, Dynamic Memory Allocation.

3.2.4 Sequential File IO

Figure F.3.3-5 shows the run time overhead measured for sequential IO.

RECORD SIZE	ACS/ADE IO TIMES					
(Bytes)	(milliseconds/read)			(milliseconds/write)		
	average	low	high	average	low	high
4	0.6/7	0.6/7	0.6/7	0.5/5	0.5/5	0.5/5
600	4.0/50	3.0/50	5.0/50	11/120	8.0/120	13/120
10000	120/130	100/130	140/130	340/280	300/280	400/280

Figure F.3.3-5, Sequential IO.

3.2.5 Tasking

Figure F.3.3-6 shows the run time overhead measured for a rendezvous between 2 tasks.

NUMBER OF PARAMETERS	PARAMETER TYPE	ACS/ADE RENDEZVOUS OVERHEAD (millisec/rendezvous)		
		average	low	high
0	-	1.8/11	1.8/11	1.8/12
1	IN	1.8/11	1.8/11	1.8/11
1	ACCESS	1.8/11	1.8/11	1.8/11
10	IN	1.8/12	1.8/12	1.9/12
10 element array	IN	1.8/11	1.8/11	1.8/11
100 element array	IN	2.0/12	1.9/12	2.0/12
1000 element array	IN	3.6/12	3.4/12	4.0/13
1k element array	INOUT	3.4/13	3.3/13	3.6/13

Figure F.3.3-6, Rendezvous Overhead.

Figure F.3.3-7 shows the run time overhead measured for dynamic task activation.

ACS/ADE TASK ACTIVATION OVERHEAD
(milliseconds/activation)

average	low	high
6.2/14	6.0/14	6.4/14

Figure F.3.3-7, Task Activation Overhead.

Figure F.3.3-8 shows the run time measured for reading, processing and writing a number of 600 bytes (NASCOM) and 10_000 bytes records (BIG BLOCKS) using tasking (PARALLEL) or not (SERIAL). Refer to the description of PAR_BIG and SER_BIG given previously for details.

	PROCESSING MODE	ACS/ADE TOTAL EXECUTION TIME (seconds)		
TWO CONTROLLERS:		average	low	high
NASCOM BLOCKS	SERIAL	3.7/NA	3.4/NA	3.9/NA
NASCOM BLOCKS	PARALLEL	4.1/NA	3.9/NA	4.4/NA
BIG BLOCKS	SERIAL	6.6/NA	6.4/NA	6.8/NA
BIG BLOCKS	PARALLEL	4.5/NA	4.3/NA	4.8/NA
ONE CONTROLLER:				
BIG BLOCKS	SERIAL	7.5/NA	7.3/NA	7.7/NA
BIG BLOCKS	PARALLEL	5.4/NA	5.2/NA	5.6/NA
NASCOM BLOCKS	SERIAL	NA/14	NA/14	NA/14
NASCOM BLOCKS	PARALLEL	NA/16	NA/16	NA/16

Figure F.3.3-8, Parallel Processing test.

Our results, obtained with the ACS, show that when separate controllers are used for the input and the output, parallelism is highest, allowing the PAR_BIG multi-tasking program to run more than 24% faster that its serial counterpart.

Excellent buffering by the OS however, makes the serial program for NASCOM blocks (SER_NB) run 10% faster that its multi-tasking counterpart.

Lack of time and numerous problems with an unfamiliar environment did not allow us to run PAR_BIG on the ADE.

3.2.6 An Interesting Math Routine

In Reference 3, it was shown that for two routines accessing an array in a stride and non-stride manner, the F77 compilers produced significantly slower code than the VAX FORTRAN and that all of the VMS Pascal compilers considered generated very inefficient code.

Our results, presented in figure F.3.3-9, show that **the VAX Ada code for this test is not as efficient as the VAX FORTRAN code (execution time for VAX FORTRAN is about half that for VAX Ada).** This result contradicts our own previous experience (see Reference 1) and the results of other groups. DEC Ada is often found to be faster than DEC FORTRAN V4.2 but we observe that DEC FORTRAN V4.3 produces significantly faster code and that the ACS optimizer can be improved. We hope that DEC Ada will benefit from the progress made for DEC FORTRAN.

ACS on VAX 11/785 CPU time
(seconds)

	50 ITER-ATIONS	100 ITER-ATIONS	150 ITER-ATIONS	200 ITER-ATIONS
STRIDING:				
VAX FORTRAN (V4.3)	0.3	1.8	6.5	15.9
VAX ADA (V1.0)	0.4	3.7	13.5	38.0
NON-STRIDING:				
VAX FORTRAN (V4.3)	0.3	2.4	8.8	25.0
VAX ADA (V1.0)	0.3	2.8	10.2	26.9

Figure F.3.3-9, Array Reference Benchmark Execution Times.

4 CONCLUSION

In general, the ACS is a reasonable system to work with. The following positive comments can be made:

- The ACS operates in a logical, easy to comprehend manner. When assistance is required, documentation on operating the ACS is complete, accurate, and easy to use. On-line help is available.

- The LRM is generously supplemented with text and examples specific to the DEC implementation.

- The ACS is well integrated into the DEC software development and run-time environment. A run-time reference manual provides practical information about internal details of the DEC implementation and how Ada interfaces to VMS and other high-level languages.

- Compilation speed is rapid enough for serious software development (at least on a VAX-11/785).

- While ACS disk space requirements (per user library unit) are high, "garbage" files, necessary to track compilation units, were fewer than on ADE and were confined to the library directory, rather than cluttering the user's working directory.

- Run-time error messages were excellent. They were generally very specific about the true nature of the problem and provided the VMS standard trace back information.

The following negative comments can be made about the ACS:

- The Ada rendezvous mechanism, which will be critical to MSOCC realtime applications, incurred relatively high overhead.

- The ACS requires large amounts of disk space to maintain a user library.

- In the single direct comparison between VAX Ada and VAX FORTRAN (ARRAY_REF), our results suggest that in spite of its overall good quality, the DEC ACS code generator can be improved.

- While the information contained in compiler error messages usually identifies the offending line of code and the nature of the error, the messages themselves tend to be verbose and poorly worded. Much effort is required to extract the information from the message.

The following positive comments can be made about the ADE:

1. The ADE features a more extensive set of tools than the ACS (e.g., a pretty printer).

2. The library manager can produce very useful cross reference reports.

3. The symbolic debugger is friendly and more mature than other systems' debuggers.

4. The ADE features an impressive number of packages (e.g., BIT_OPS to alleviate the lack of representation clauses, CURRENT_EXCEPTION to help determine the origin of an exception) that would help alleviate some of the problems we mentioned.

5. Overall, the ADE generated less efficient code than the ACS but in a few cases, when the difference in CPU speed is accounted for, the ADE generated code of equal or better quality.

The following negative comments can be made about the ADE:

1. Run-time error messages were terrible. Frequently, system limits are exceeded during program elaboration. When this happens, the user is either presented with "Unhandled exception in library unit prog", or "Constraint error in unit main", and no additional information.

2. The compiler required a pragma or a compile switch to explicitly declare a procedure to be the main program. The concept is not part of the LRM and should not be necessary.

3. Compilation times were very slow, even considering the fact that the MV/4000 is only a 0.6 MIPS machine.

4. PAR_NB contained code which assigned an array to an array with an explicit type conversion (the arrays were declared as different types). The DG compiler generated bad code which caused the program to hard abort directly to the operating system with no Ada exception raised. PAR_NB was recoded to avoid the type conversion.

5. PAR_BIG never ran successfully on the MV/4000. An exception was raised the first time that a read was attempted on its input file. The reason for the exception was not apparent. SER_BIG did not have any problems reading the file. PAR_BIG worked correctly on the VAX.

6. The ADE doesn't support storage of 8-bit integers. It uses 32-bits for all integer variables, ignoring length representation clauses. In order to compare ADE IO benchmark results to ACS results, programs were modified on the MV/4000 to ensure that buffers were the same number of bytes.

7. There is no ADE compiler switch to turn off optimization. Such a switch is frequently necessary when working with symbolic debuggers and would have been useful in the benchmarking process.

8. The user's guide was rather thin and did not provide much insight into the ADE implementation of the Ada language.

9. The ADE LRM documentation did not include any ADE specific description or examples.

10. Some of the library files that ADE needs to configure compilation units must reside in the users working directory rather than in the library directory. Users have a hard enough time keeping their directories free of their own "garbage" files without also having to worry about the ADE's. The names generated for the ADE files have very long and arcane embedded number sequences, making them unwieldy to deal with on an individual basis.

11. The ADE makes extravagant use of disk space.

12. The MV/4000 text editor (SED) didn't seem to us as friendly as the VAX's (EDT). However, a colleague demonstrated a very impressive Ada frame driven editor the he built using SLATE's macro capability.

Overall, the ADE is usable for investigating the Ada language but many improvements are needed before it can be used as a production compiler.

4.1 For More Information

Two reports, available from the authors, document the suite and the results of the comparison between DEC's ACS and DG's ADE:

- An Evaluation Suite for Ada Compilers, Century Computing, Inc., Revision A, March 1986.

- A Comparison of the DEC Ada Compilation System and the DG Ada Development Environment, Century Computing, Inc., Revision A, March 1986.

The source code for the suite and the RUNOFF source file for the reports are also available from the authors on a VMS BACKUP format tape.

5 BIBLIOGRAPHY

1. Evaluation of Ada in the MSOCC Environment, Final Report, Century Computing, Inc., July 31, 1985.

2. Where are the Optimizing Compilers?, Wolfe/Macke, SIGPLAN Notices, V20, #11, November, 1985.

3. Dhrystone: A Synthetic Systems Programming Benchmark, Weicker, Communications of the ACM, Volume 27, Number 10, October, 1984.

4. Evaluating the Performance Efficiency of Ada Compilers, Bassman et al, Proceedings of the Washington Ada Symposium, ACM, 1985.

Randy Wilke is a senior member of the technical staff at Century Computing Inc. where he has been working since 1981. He received a Bachelor of Science in Computer Science from the University of Southern California in 1976.

Daniel Roy is a senior member of the technical staff at Century Computing Inc. where he has been working since 1983. He received the Diplome d'Ingenieur Electronicien (MSEE) from ENSEA in 1973 and the Diplome d'Etudes Approfondies en Informatique (MSCS) from the University of Paris VI in 1975.

Authors' current address:
Century Computing, Inc., 1100 West street, Laurel, Md., 20707.
Tel: (301) 953-3330.

PARANOIA . ADA :

A DIAGNOSTIC PROGRAM TO EVALUATE ADA FLOATING-POINT ARITHMETIC

May 12, 1985

Chris Hjermstad
Package_Architects, Inc.
8950 Villa La Jolla Drive
Suite 1200
La Jolla, California 92037
(619) 587-1815

INTRODUCTION

Programmers have traditionally approached floating-point arithmetic with great trepidation. Brown and Feldman in their landmark paper on model numbers call floating-point arithmetic the "*bete noire*" (black beast) of computing. Programmers are haunted by the suspicion that floating-point calculations harbor hidden errors. This resistance stems, at least partly, from the variety of inconsistent floating-point representations implemented over the years by different computer manufacturers.

In many respects, the programming language Ada* is not so much a breakthrough in technology as it is an evolutionary melding of many advancements achieved by computer science research during the 1970s. This is certainly the case with respect to Ada's treatment of floating-point arithmetic. Ada explicitly adheres to concepts of environmental inquiry initially proposed by Naur in 1967 and of model number parameterization advanced by Brown and Feldman in 1980 and formalized by Brown in 1981. Following these precepts, Ada encourages the development of safe, transportable numerical programs. This paper traces major historical efforts to establish effective standards for floating-point arithmetic. It describes previously developed programs written in languages such as FORTRAN and BASIC which partially undertake the testing of conformance to such standards. It provides results obtained from a contemporary program, Paranoia.Ada, which tests various aspects of floating-point arithmetic in the context of the Ada programming language.

SPECIFICATION OF FLOATING-POINT ARITHMETIC

The last two decades have witnessed efforts within the computer science community to establish floating-point arithmetic standards. These efforts have been primarily motivated by a desire to perform consistent arithmetic in a common transportable programming language across many different computing environments and hardware architectures. Naur, writing in 1967, introduced the concept of an "environmental inquiry" as a means of ascertaining the arithmetic characteristics of a computing environment. His ideas where incorporated into the ALGOL 68 language and are reflected in the "attribute" feature of Ada.

The International Federation for Information Processing (IFIP) Working Group 2.5 (Mathematical Software) introduced the concept of floating-point parameters as a means of determining the characteristics of a specific programming environment's floating-point arithmetic implementation. The design of FORTRAN 77 provided access to such floating-point parameters.[1]

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

BROWN-FELDMAN CONTRIBUTIONS

More recently, Brown and Feldman, using model number theory, further specified floating-point parameterization. They defined a generalized standard representation of floating-point numbers independent of underlying machine architecture. Their landmark work resulted in precise definitions for floating-point arithmetic based on model numbers and model intervals. They established rigorous theorems concerning the dependability of computational results derived from operations that adhered to the basic model definitions.[2] They defined a standard model number representation as:

$x = b^{e} f$, where

b is the specified radix,

e is an integer exponent of specified range, and

f is the significand expressed as a base-b digit.

They identified seven model parameters as necessary to the specification of a floating-point arithmetic implementation. Four parameters consist of basic integer values:

BASE	b
PRECISION	p
MINIMUM EXPONENT	emin
MAXIMUM EXPONENT	emax

Three additional parameters consist of floating-point values derivable from the basic parameters:

MAXIMUM RELATIVE SPACING	Epsilon = b^{1-p}
SMALLEST POSITIVE NUMBER	Sigma = b^{emin-1}
LARGEST NUMBER	Lambda = $b^{emax}(1-b^{-p})$

CURRENT IEEE STANDARDIZATION EFFORTS

Two committees within the IEEE are working to further refine and extend the Brown-Feldman model of floating-point arithmetic. Committee P754 is developing a detailed specification to be applied to computers employing a binary representation. P854 is developing a compatible super-set specification that is both radix and word length independent. The objective of both committees is to establish additional environmental rules which will precisely define the outcome of all floating-point operations. Such rules are intended to eliminate all implementation-dependent or ambiguous circumstances with particular emphasis on consistent treatment of error conditions.[3]

As an example of this focus, both IEEE draft specifications require the implementation of at least the five following exception conditions:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact result

ADA FLOATING-POINT PARAMETERIZATION

The design of floating-point arithmetic in the Ada programming language explicitly complies with the Brown-Feldman model. It requires a limited, conservative interpretation of the Brown-Feldman parameters. Ada assumes a binary representation and arbitrarily assigns values to the other parameters based on the elemental precision specification of DIGITS in a real object type definition. Although the minimal Ada model number parameter values frequently result in an artificially limited precision range, they do encourage portability, predictability and understandability.

Ada also allows for the specification of implementation-dependent "safe number" values. Such safe numbers permit additional latitude in the programming of numerically sophisticated procedures requiring greater exploitation of the complete underlying hardware architecture. A comparison between the Brown-Feldman parameters against Ada attributes relating to both the required model number values and implementation-dependent values shows a close mapping:

Brown-Feldman Parameter	Model Number Attribute	Implementation-Dependent Attribute
b	2 (BY DEFINITION)	T'MACHINE_RADIX
p	T'MANTISSA (FUNCTION OF T'DIGITS)	T'MACHINE_MANTISSA
emax	T'EMAX (4*T'MANTISSA)	T'SAFE_EMAX T'MACHINE_EMAX
emin	- T'EMAX (SYMMETRICAL RANGE)	T'MACHINE_EMIN
Epsilon	T'EPSILON (2.0**(1 - T'MANTISSA))	(Determined by Paranoia.Ada)
Sigma	T'SMALL (2.0**(-T'EMAX - 1))	T'SAFE_SMALL
Lambda	T'LARGE (2**T'EMAX * (1.0 - 2.0**(-T'MANTISSA)))	T'SAFE_LARGE

HISTORICAL FLOATING-POINT ARITHMETIC TESTS

A number of computer programs have been written in the last several years which evaluate the quality of floating-point arithmetic implementations. One such program is MACHAR written by Cody in 1979 and published in the classic reference, *Software Manual for the Elementary Functions.*[4] MACHAR, coded in FORTRAN 77, determines thirteen characteristics of a floating-point arithmetic implementation such as radix, precision, rounding phenomenon, underflow threshold and overflow threshold.

Another notable effort is the Arithmetic Unit Test Program developed by Schryer in 1979. Results from the execution of this program were reported in the seminal Brown and Feldman paper "Environmental Parameters and Basic Functions for Floating-Point Computation".[5] Schryer's test program was also coded in FORTRAN 77 and calculates the seven Brown-Feldman model parameters. The program was used to test Cray-1, IBM

370, DEC VAX, Honeywell 6000 and Interdata 8/32 computers in support of Brown and Feldman's research.

RECENT FLOATING-POINT ARITHMETIC TESTS

More recently, two members of the IEEE floating-point standardization committees have written programs that perform even more sophisticated evaluations of floating-point arithmetic implementations. Karpinsky's 1985 article, "Paranoia: A Floating-Point Benchmark" describes the program Paranoia written by University of California, Berkeley Professor W. M. Kahan.[6] The article includes both Pascal and BASIC source code listings of Guard, a subset version of the full Paranoia. Kahan's original Paranoia is written in BASIC for the IBM (Intel 8088/8087) Personal Computer. It has also been translated into FORTRAN, Pascal and "C" for execution on DEC VAX and Sun Microcomputer (Motorola 68000) architectures.[7]

ADA IMPLEMENTATION OF PARANOIA

In conjunction with its Ada evaluation activities, Package_Architects, Inc. has converted the original Paranoia program to Ada. This converted program is called Paranoia.Ada. Paranoia.Ada determines the floating-point characteristics of the hardware supporting an Ada implementation. It also evaluates the accuracy, precision and reliability of the basic, predefined Ada arithmetic operations. The program identifies errors in floating-point computations and provides a report summarizing the overall quality and acceptability of the floating-point computational capability.

Paranoia.Ada performs specific diagnostic tests related to the following aspects of floating-point arithmetic:

- Determination of correct mathematical operations on small integral values.
- Calculation of radix, precision and Epsilon parameters.
- Determination of normalization with respect to subtraction operations.
- Determination of guard digits on subtraction, multiplication and division operations.
- Determination of rounding phenomenon (e.g. chopped, rounded or rounded to even) on addition, subtraction, multiplication and division.
- Determination of commutative multiplication properties.
- Determination of underflow threshold values.

- Determination of rounding phenomenon on floating-point to integer conversion operations.
- Determination of overflow threshold values.
- Evaluation of integer power arithmetic.
- Evaluation of division by zero arithmetic.

Paranoia.Ada takes significant advantage of several advanced features of Ada. The program relies on the Ada exception feature to detect and respond to error conditions with less disruption to processing than occurs with conventional BASIC or Pascal mechanisms. The program has been architecturally redesigned into forty-six separately compiled units and consists of approximately twenty-five hundred semi-colon terminated Ada statements. The program is implemented as a generic and is instantiated through the specification of a DIGITS parameter or by reference to a predefined FLOAT_TYPE.

Because a number of validated Ada compilers do not provide the mathematical functions required by the Paranoia algorithms, Paranoia.Ada contains a partial mathematics library based on the Cody-Waite algorithms. The program can either use the mathematics library provided by the compiler being tested or use its own independent library for test calculations.

The program also includes a utility package called STOP_WATCH which provides timing data related to test execution. The program measures the amount of CPU time required to perform the floating-point diagnostic tests and the amount of time required to generate the resulting output report.

PARANOIA.ADA DIAGNOSTIC EVALUATIONS

Paranoia.Ada replicates the test algorithms implemented in the original BASIC language version and adheres to the evaluation criteria established by Professor Kahan. Paranoia.Ada classifies errors detected in the course of its diagnosis into four categories. Ranked according to increasing levels of severity, the error categories consist of flaws, defects, serious defects and failures. Examples of errors associated with each category are as follows:

Flaws:	Comparison anomalies such as: X /= -(-(X) or, X /= Y but X - Y = 0.

	Range imbalance between overflow threshold and underflow threshold.
Defects:	Comparison anomalies such as: $Z{*}{*}I \mathrel{/}= Z_1{*}Z_2{*}Z_3....{*}Z_I$.
	Erroneously raised numeric errors.
	An imbalance between the underflow threshold and Epsilon.
	Multiplication and subtraction operations yield inconsistent underflow thresholds.
Serious Defects:	Absence of division by zero protection.
	Absence of guard digits.
	Underflow or overflow conditions not accompanied by corresponding numeric errors.
Failures:	Outright arithmetic errors such as: $2 + 2 = 5$.
	Non-normalized subtraction.
	Erroneous guard digits.
	Underflow to negative number.
	Accuracy deterioration approaching underflow.

Paranoia.Ada maintains a record of the errors encountered in the course of its execution. In its summary report, the program generates an overall evaluation of the tested floating-point implementation. Using IEEE Standards P754 and P854 as criteria, the program rates the diagnosed arithmetic in terms of one of the following comments:

- The arithmetic diagnosed appears excellent.
- The arithmetic diagnosed seems satisfactory.
- The arithmetic diagnosed seems satisfactory though flawed.
- The arithmetic diagnosed may be acceptable despite inconvenient defects.
- The arithmetic diagnosed has unacceptable serious defects.
- A fatal failure may have spoiled this program's subsequent diagnoses.

EXECUTION OF PARANOIA.ADA AGAINST DEC ACS

Paranoia.Ada has been run extensively against the Digital Equipment Corporation (DEC) Ada Compilation System (ACS) hosted on a VAX 785 computer. The VAX architecture provides a rich and powerful floating-point arithmetic capability. The VAX supports four floating-point representations. These four representations are available through the Ada package SYSTEM pre-defined floating-point types F_FLOAT, D_FLOAT, G_FLOAT and H_FLOAT. F_FLOAT is a 32 bit representation, D_FLOAT and G_FLOAT are alternative 64 bit representations (selectable by a PRAGMA directive), and H_FLOAT is a 128 bit representation.

The DEC ACS also provides for three pre-defined floating-point types in package STANDARD. The compiler maps each of these types -- FLOAT, LONG_FLOAT, and LONG_LONG_FLOAT -- into the respective machine representation types F_FLOAT, D_FLOAT or G_FLOAT, and H_FLOAT. Paranoia.Ada has been run against all seven of these pre-defined types as well as a user-defined type of SYSTEM.MAX_DIGITS. SYSTEM.MAX_DIGITS forces the compiler to use the H_FLOAT representation. Sample output reports from D_FLOAT, G_FLOAT, H_FLOAT, and SYSTEM.MAX_DIGITS test runs are supplied as attachments.

DIAGNOSTIC ANALYSIS

Paranoia.Ada provides a consistent diagnosis of the eight tested floating-point representations. The values calculated by the Paranoia.Ada algorithms match the values reported by queries to corresponding Ada attributes. The program detects a similar set of errors on all eight representations as well. One flaw and one serious defect pertaining to underflow phenomena were discovered for each of the representations. The flaw involves an inconsistency between comparison results and arithmetic results with numerical values at or very close to the underflow threshold. The serious defect concerns the absence of a numeric error when subtraction operations on such small numbers result in underflow.

This specific circumstance is addressed by the IEEE standards. The DEC VAX implementation appears to result in an underflow to zero but without a numeric error being raised. The IEEE standards require that the underflow result be a non-normalized "tiny" number accompanied by an exception.

Paranoia.Ada uncovers a second serious defect in the D_FLOAT floating-point representation. In the VAX architecture, D_FLOAT representation is an extension of the single-precision F_FLOAT representation. (G_FLOAT is the true double-precision representation.) D_FLOAT has the same exponent range as F_FLOAT but uses an additional 32 bits of storage to allow greater precision in the significand. This allocation violates a requirement of the IEEE specification for balance between Epsilon and Sigma. In Paranoia.Ada terms, Epsilon equates to a calculated unit in the last place value and Sigma is the calculated underflow threshold value.

TIMING RESULTS

Execution and compilation timing data for each of the eight various DEC ACS floating-point representations are presented in Table 1. Execution times are also graphically depicted in Figure 1. (Since these data represent only a single sample for each type, caution is advised against drawing unjustified general conclusions.) Report generation times appear relatively consistent and provide a basis of comparison for the execution time differences. The execution times appear to increase as a function of the amount of precision provided by each type. Within the same precision, STANDARD pre-defined types seem to take longer to execute than SYSTEM pre-defined types.

Compilation times for the seven pre-defined types are also relatively constant. For these types, the DEC ACS compiles Paranoia.Ada at a rate of approximately six hundred statements per minute. The compiler generates the SYSTEM.MAX_DIGITS version of the program at a slightly slower rate.

SIGNIFICANCE OF RESULTS

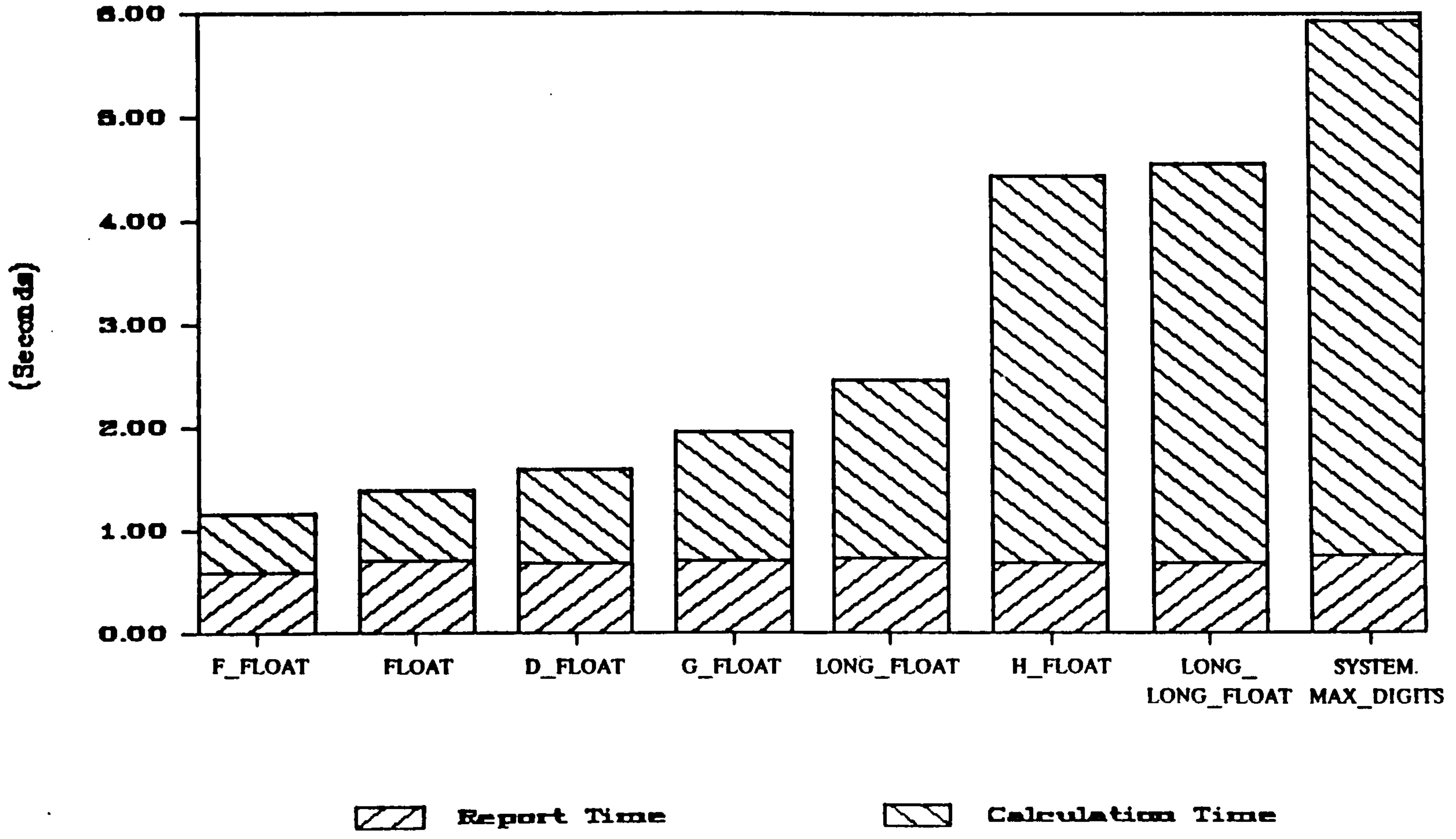
Owing to the sophistication of its diagnostic algorithms, Paranoia.Ada places heavy demands on the floating-point capabilities of an Ada compiler. The successful compilation and execution of a program as numerically complex and devious as Paranoia.Ada is a significant demonstration of a compiler's maturity, robustness and completeness. Paranoia.Ada is a practical exploration of Ada's floating-point capabilities. It tests the fidelity of an Ada implementation to the concept of model numbers, assesses the dependability of the arithmetic, and reveals Ada's suitability as an engine for further serious numerical computations. Paranoia.Ada, itself being a computationally intensive program, establishes the appropriateness of Ada as a medium for numerically demanding applications.

TABLE 1: Paranoia.Ada Compilation, Diagnostic Calculation and Report Generation Times

Type	F_FLOAT	FLOAT	D_FLOAT	G_FLOAT	LONG_FLOAT	H_FLOAT	LONG_ LONG_FLOAT	SYSTEM. MAX_DIGITS
Type'SIZE	32 Bits	32 Bits	64 Bits	64 Bits	64 Bits	128 Bits	128 Bits	128 Bits
Compilation Time	04:18.12	04:20.87	04:20.51	04:19.85	04:20.32	04:22.26	04:22.56	04:37.22
Diagnostic Calculation Time	00:00.57	00:00.68	00:00.93	00:01.26	00:1.72	00:03.75	00:03.86	00:04.42
Report Generation Time	00:00.60	00:00.71	00:00.68	00:00.71	00:00.74	00:00:69	00:00.70	00:00.71

Note: Times In minutes and seconds.

FIGURE 1: Paranoia.Ada Execution Time Comparison



CONCLUSION

Many essential software functions in the mission critical computer resource application domain depend on floating-point arithmetic. Numerically intensive functions associated with the Space Station project, such as ephemeris generation or the implementation of Kalman filters, are likely to employ the floating-point facilities of Ada. Paranoia.Ada appears to be a valuable program to insure that Ada environments and their underlying hardware exhibit the precision and correctness required to satisfy mission computational requirements.

As a diagnostic tool, Paranoia.Ada reveals many essential characteristics of an Ada floating-point implementation. Equipped with such knowledge, programmers need not tremble before the "black beast" of floating-point computation.

REFERENCES

1. W. J. Cody, "Floating-point Parameters, Models and Standards," in *The Relationship Between Numerical Computation and Programming Languages*, J.K. Reid, ed., North-Holland Publishing Co., Amsterdam, 1982, pp. 51-65.

2. W. S. Brown, "A Simple But Realistic Model of Floating-point Computation," *ACM Transactions on Mathematical Software*, Vol. 7, No. 4, December 1981, pp. 445-480.

3. W. J. Cody, W. Kahan, et. al. "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic," *IEEE Micro*, August 1984, pp. 86-100.

4. W. J. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.

5. W. S. Brown and S. I. Feldman, "Environmental Parameters and Basic Functions for Floating-point Computation", *ACM Transactions on Mathematical Software*, Vol. 6, No. 4, December 1980, pp. 510-523.

6. R. Karpinsky, "Paranoia: A Floating-Point Benchmark", *BYTE*, Vol. 10, No. 2, February 1985, pp. 223-235.

7. BASIC, FORTRAN, Pascal and "C" Paranoia source code is available from Mr. Richard Karpinsky, IEEE P854 Mailings, U-76, University of California, San Francisco, San Francisco, CA 94143.

N89-16362

PARANOIA . ADA :

SAMPLE OUTPUT REPORTS

```
------------------------------------------------------------

                    Paranoia.Ada (R)

                 A Diagnostic Benchmark

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Paranoia.Ada is a program to diagnose floating-point arithmetic
in the context of the Ada(*) programming language. The program
evaluates the quality of a floating-point arithmetic implementa-
tion with respect to the proposed IEEE Standards P754 and P854.

Paranoia.Ada is derived from the original BASIC programming
language version of Paranoia developed and copyrighted by
Professor W.M. Kahan of the University of California, Berkeley.

The BASIC Paranoia program is described in "Paranoia: A Floating-
Point Benchmark", by Richard Karpinsky, Byte Magazine, Vol. 10,
No. 2, February 1985, pp. 223-235.

Paranoia.Ada replicates in Ada the test algorithms originally
implemented in BASIC and adheres to the evaluation criteria
established by Kahan.

Paranoia.Ada incorporates a major structural redesign and
employs applicable Ada architectural and stylistic features.

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

(C) Package_Architects, Inc. 1986

------------------------------------------------------------
```

```
Paranoia Report for type D_FLOAT                19:38:42  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)

SYSTEM_NAME: VAX_VMS

Small Integral Values Test

Radix, Precision, and Closest Relative Separation Test

Normalized Subtraction Test

Guard Digit on Subtraction Test

Guard Digit on Multiplication Test

Guard Digit on Division Test

Rounding for Addition/Subtraction Test

Rounding for Multiplication Test

Rounding for Division Test

Rounding Sticky Bit Test

Commutative Multiplication Test

Underflow Test
    FLAW:
        Comparison says X /= Z, and yet X - Z = 0.0
        X = 4.04076183095161331E-39
        Z = 2.93873587705571888E-39
    SERIOUS DEFECT:
        Exception NUMERIC_ERROR was NOT raised to report
        underflow for Y := X - Z
        Confusion will be caused when innocent statements
        like:
            if (x=z)
            then ...
            else ... (f(x)-f(z))/(x-z) ...
        encounter division by zero although actually
        X / Z = 1.0 + 3.7500000000000000E-01
    SERIOUS DEFECT:
        Range is too narrow, U1 ** 4 underflows

Conversion Rounding Test

Overflow Test

Integer Power Test

Division by Zero Test
```

```
Paranoia Summary for type D_FLOAT              19:38:42  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


SERIOUS DEFECTS discovered                    2
FLAWS discovered                              1

Small Integral Values -1.0, 0.0, 0.5, 1.0, 2.0,
                       3.0, 4.0, 5.0, 8.0, 9.0,
                       24.0, 27.0, 32.0, and 240.0 are o.k.

Radix, Precision, and Closest Relative Separation

  Ada Implementation Attributes
    D_FLOAT'MACHINE_RADIX                      2
    D_FLOAT'MACHINE_MANTISSA                  56 bits
    D_FLOAT'MANTISSA                          31 bits
    D_FLOAT'DIGITS                             9 decimal digits
    D_FLOAT'EPSILON
         9.31322574615478852E-10

  Calculated Values
    Radix                                      2
    Precision                                 56 digits of Radix
    U1 [1.0 - nextafter(1.0, 0.0)]
         1.38777878078144578E-17
    U2 [nextafter(1.0, 2.0) - 1.0]
         2.77555756156289914E-17

Subtraction is normalized

Guard Digits

    Subtraction:    has guard digit
    Multiplication: has guard digit
    Division:       has guard digit

Rounding

  Ada Implementation Attributes
    D_FLOAT'MACHINE_ROUNDS                   TRUE

  Calculated Values
    Addition/Subtraction: appears to be correctly rounded
    Multiplication:       appears to be correctly rounded
    Division:             appears to be correctly rounded
    Sticky bit;           used incorrectly or not at all

Multiply commutes correctly for 20 pairs

Underflow

  Ada Implementation Attributes
    D_FLOAT'SMALL
```

```
Paranoia Summary for type D_FLOAT              19:38:42  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


          2.3509887016445750E-38
      D_FLOAT'SAFE_SMALL
          2.9387358770557188E-39

    Calculated Values
      E0 - smallest positive number
          2.9387358770557188E-39
      U0 - underflow threshold
          2.9387358770557188E-39

Conversion from +-x.5 to INTEGER rounds FROM_ZERO

Overflow

    Ada Implementation Attributes
      D_FLOAT'MACHINE_OVERFLOWS            TRUE
      D_FLOAT'LARGE
          2.1267647922655134E+37
      D_FLOAT'SAFE_LARGE
          1.7014118338124107E+38
      D_FLOAT'LAST
          1.7014118346046923E+38

    Calculated Values
      V0 - overflow saturation
          1.7014118346046923E+38
      V  - overflow threshold
          1.7014118346046923E+38

Integer Powers were calculated correctly

Division by 0.0 handled correctly


With respect to the proposed IEEE standards P754 and P854:
    The arithmetic diagnosed has unacceptable serious defects


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Paranoia calculations elapsed time         0.9300 seconds
Paranoia report elapsed time               0.6800 seconds
Paranoia Report for type G_FLOAT                 20:05:18  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)

SYSTEM_NAME: VAX_VMS

Small Integral Values Test

Radix, Precision, and Closest Relative Separation Test

Normalized Subtraction Test

Guard Digit on Subtraction Test

Guard Digit on Multiplication Test

Guard Digit on Division Test

Rounding for Addition/Subtraction Test

Rounding for Multiplication Test

Rounding for Division Test

Rounding Sticky Bit Test

Commutative Multiplication Test

Underflow Test
    FLAW:
        Comparison says X /= Z, and yet X - Z = 0.0
        X = 7.6486913886185O5E-309
        Z = 5.562684646268003E-309
    SERIOUS DEFECT:
        Exception NUMERIC_ERROR was NOT raised to report
        underflow for Y := X - Z
        Confusion will be caused when innocent statements
        like:
            if (x=z)
            then ...
            else ... (f(x)-f(z))/(x-z) ...
        encounter division by zero although actually
        X / Z = 1.0 + 3.750000000000000E-01

Conversion Rounding Test

Overflow Test

Integer Power Test

Division by Zero Test
```

```
Paranoia Summary for type G_FLOAT            20:05:18  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


SERIOUS DEFECTS discovered                    1
FLAWS discovered                              1

Small Integral Values -1.0, 0.0, 0.5, 1.0, 2.0,
                       3.0, 4.0, 5.0, 8.0, 9.0,
                       24.0, 27.0, 32.0, and 240.0 are o.k.

Radix, Precision, and Closest Relative Separation

  Ada Implementation Attributes
    G_FLOAT'MACHINE_RADIX                      2
    G_FLOAT'MACHINE_MANTISSA                  53 bits
    G_FLOAT'MANTISSA                          51 bits
    G_FLOAT'DIGITS                            15 decimal digits
    G_FLOAT'EPSILON
         8.881784197001252E-16

  Calculated Values
    Radix                                      2
    Precision                                 53 digits of Radix
    U1 [1.0 - nextafter(1.0, 0.0)]
         1.110223024625157E-16
    U2 [nextafter(1.0, 2.0) - 1.0]
         2.220446049250313E-16

Subtraction is normalized

Guard Digits

    Subtraction:    has guard digit
    Multiplication: has guard digit
    Division:       has guard digit

Rounding

  Ada Implementation Attributes
    G_FLOAT'MACHINE_ROUNDS                   TRUE

  Calculated Values
    Addition/Subtraction: appears to be correctly rounded
    Multiplication:       appears to be correctly rounded
    Division:             appears to be correctly rounded
    Sticky bit;           used incorrectly or not at all

Multiply commutes correctly for 20 pairs

Underflow

  Ada Implementation Attributes
    G_FLOAT'SMALL
```

```
Paranoia Summary for type G_FLOAT                20:05:18  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


          1.944692274331607E-62
     G_FLOAT'SAFE_SMALL
          5.562684646268003E-309

   Calculated Values
     E0 - smallest positive number
          5.562684646268003E-309
     U0 - underflow threshold
          5.562684646268003E-309

Conversion from +-x.5 to INTEGER rounds FROM_ZERO

Overflow

   Ada Implementation Attributes
     G_FLOAT'MACHINE_OVERFLOWS                TRUE
     G_FLOAT'LARGE
          2.571100870814383E+61
     G_FLOAT'SAFE_LARGE
          8.988465674311576E+307
     G_FLOAT'LAST
          8.988465674311579E+307

   Calculated Values
     V0 - overflow saturation
          8.988465674311579E+307
     V  - overflow threshold
          8.988465674311579E+307

Integer Powers were calculated correctly

Division by 0.0 handled correctly


With respect to the proposed IEEE standards P754 and P854:
    The arithmetic diagnosed has unacceptable serious defects


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Paranoia calculations elapsed time        1.2600 seconds
Paranoia report elapsed time              0.7100 seconds
Paranoia Report for type H_FLOAT              20:47:00  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


SYSTEM_NAME: VAX_VMS

Small Integral Values Test

Radix, Precision, and Closest Relative Separation Test

Normalized Subtraction Test

Guard Digit on Subtraction Test

Guard Digit on Multiplication Test

Guard Digit on Division Test

Rounding for Addition/Subtraction Test

Rounding for Multiplication Test

Rounding for Division Test

Rounding Sticky Bit Test

Commutative Multiplication Test

Underflow Test
    FLAW:
        Comparison says X /= Z, and yet X - Z = 0.0
        X = 1.155722955444782142777795499704352E-4932
        Z = 8.405257857780233765656694543304382E-4933
    SERIOUS DEFECT:
        Exception NUMERIC_ERROR was NOT raised to report
        underflow for Y := X - Z
        Confusion will be caused when innocent statements
        like:
            if (x=z)
            then ...
            else ... (f(x)-f(z))/(x-z) ...
        encounter division by zero although actually
        X / Z = 1.0 + 3.750000000000000000000000000000000E-01

Conversion Rounding Test

Overflow Test

Integer Power Test

Division by Zero Test
```

```
Paranoia Summary for type H_FLOAT                 20:47:00  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


SERIOUS DEFECTS discovered                      1
FLAWS discovered                                1

Small Integral Values -1.0, 0.0, 0.5, 1.0, 2.0,
                       3.0, 4.0, 5.0, 8.0, 9.0,
                       24.0, 27.0, 32.0, and 240.0 are o.k.

Radix, Precision, and Closest Relative Separation

  Ada Implementation Attributes
    H_FLOAT'MACHINE_RADIX                         2
    H_FLOAT'MACHINE_MANTISSA                    113 bits
    H_FLOAT'MANTISSA                            111 bits
    H_FLOAT'DIGITS                               33 decimal digits
    H_FLOAT'EPSILON
         7.703719777548943412223911770339709E-34

  Calculated Values
    Radix                                         2
    Precision                                   113 digits of Radix
    U1 [1.0 - nextafter(1.0, 0.0)]
         9.629649721936179265279889712924637E-35
    U2 [nextafter(1.0, 2.0) - 1.0]
         1.925929944387235853055977942584927E-34

Subtraction is normalized

Guard Digits

    Subtraction:    has guard digit
    Multiplication: has guard digit
    Division:       has guard digit

Rounding

  Ada Implementation Attributes
    H_FLOAT'MACHINE_ROUNDS                      TRUE

  Calculated Values
    Addition/Subtraction: appears to be correctly rounded
    Multiplication:       appears to be correctly rounded
    Division:             appears to be correctly rounded
    Sticky bit;           used incorrectly or not at all

Multiply commutes correctly for 20 pairs

Underflow

  Ada Implementation Attributes
    H_FLOAT'SMALL
```

```
Paranoia Summary for type H_FLOAT              20:47:00  22-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)

        1.1006568214637918210934318020936O5E-134
    H_FLOAT'SAFE_SMALL
        8.405257857780233765656694543304382E-4933

  Calculated Values
    E0 - smallest positive number
        8.405257857780233765656694543304382E-4933
    U0 - underflow threshold
        8.405257857780233765656694543304382E-4933

Conversion from +-x.5 to INTEGER rounds FROM_ZERO

Overflow

  Ada Implementation Attributes
    H_FLOAT'MACHINE_OVERFLOWS               TRUE
    H_FLOAT'LARGE
        4.542742026847543065933273799300027E+133
    H_FLOAT'SAFE_LARGE
        5.948657476786158825428796633140033E+4931
    H_FLOAT'LAST
        5.948657476786158825428796633140035E+4931

  Calculated Values
    V0 - overflow saturation
        5.948657476786158825428796633140035E+4931
    V  - overflow threshold
        5.948657476786158825428796633140035E+4931

Integer Powers were calculated correctly

Division by 0.0 handled correctly


With respect to the proposed IEEE standards P754 and P854:
    The arithmetic diagnosed has unacceptable serious defects


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Paranoia calculations elapsed time       3.7500 seconds
Paranoia report elapsed time             0.6900 seconds
Paranoia Report for type MAX_DIGITS              20:09:27  28-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)

SYSTEM_NAME: VAX_VMS

Small Integral Values Test

Radix, Precision, and Closest Relative Separation Test

Normalized Subtraction Test

Guard Digit on Subtraction Test

Guard Digit on Multiplication Test

Guard Digit on Division Test

Rounding for Addition/Subtraction Test

Rounding for Multiplication Test

Rounding for Division Test

Rounding Sticky Bit Test

Commutative Multiplication Test

Underflow Test
    FLAW:
        Comparison says X /= Z, and yet X - Z = 0.0
        X = 1.1557229554447821427777954997043352E-4932
        Z = 8.4052578577802337656566945433043822E-4933
    SERIOUS DEFECT:
        Exception NUMERIC_ERROR was NOT raised to report
        underflow for Y := X - Z
        Confusion will be caused when innocent statements
        like:
            if (x=z)
            then ...
            else ... (f(x)-f(z))/(x-z) ...
        encounter division by zero although actually
        X / Z = 1.0 + 3.7500000000000000000000000000000000E-01

Conversion Rounding Test

Overflow Test

Integer Power Test

Division by Zero Test
```

```
Paranoia Summary for type MAX_DIGITS          20:09:27  28-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


SERIOUS DEFECTS discovered                     1
FLAWS discovered                               1

Small Integral Values -1.0, 0.0, 0.5, 1.0, 2.0,
                       3.0, 4.0, 5.0, 8.0, 9.0,
                       24.0, 27.0, 32.0, and 240.0 are o.k.

Radix, Precision, and Closest Relative Separation

  Ada Implementation Attributes
    MAX_DIGITS'MACHINE_RADIX                    2
    MAX_DIGITS'MACHINE_MANTISSA               113 bits
    MAX_DIGITS'MANTISSA                       111 bits
    MAX_DIGITS'DIGITS                          33 decimal digits
    MAX_DIGITS'EPSILON
         7.703719777548943412223911770339709E-34

  Calculated Values
    Radix                                       2
    Precision                                 113 digits of Radix
    U1 [1.0 - nextafter(1.0, 0.0)]
         9.629649721936179265279889712924637E-35
    U2 [nextafter(1.0, 2.0) - 1.0]
         1.925929944387235853055977942584927E-34

Subtraction is normalized

Guard Digits

    Subtraction:    has guard digit
    Multiplication: has guard digit
    Division:       has guard digit

Rounding

  Ada Implementation Attributes
    MAX_DIGITS'MACHINE_ROUNDS                TRUE

  Calculated Values
    Addition/Subtraction: appears to be correctly rounded
    Multiplication:       appears to be correctly rounded
    Division:             appears to be correctly rounded
    Sticky bit;           used incorrectly or not at all

Multiply commutes correctly for 20 pairs

Underflow

  Ada Implementation Attributes
    MAX_DIGITS'SMALL
```

```
Paranoia Summary for type MAX_DIGITS          20:09:27  28-Apr-1986
DEC ACS 1.0-7 VAX/785 (VMS 4.2)


         1.1006568214637918210934318020936O5E-134
    MAX_DIGITS'SAFE_SMALL
         8.4052578577802337656566945433O4382E-4933

  Calculated Values
    EO - smallest positive number
         8.4052578577802337656566945433O4382E-4933
    UO - underflow threshold
         8.4052578577802337656566945433O4382E-4933

Conversion from +-x.5 to INTEGER rounds FROM_ZERO

Overflow

  Ada Implementation Attributes
    MAX_DIGITS'MACHINE_OVERFLOWS          TRUE
    MAX_DIGITS'LARGE
         4.5427420268475430659332737993000027E+133
    MAX_DIGITS'SAFE_LARGE
         5.9486574767861588254287966331400033E+4931
    MAX_DIGITS'LAST
         5.9486574767861588254287966331400035E+4931

  Calculated Values
    VO - overflow saturation
         5.9486574767861588254287966331400035E+4931
    V  - overflow threshold
         5.9486574767861588254287966331400035E+4931

Integer Powers were calculated correctly

Division by 0.0 handled correctly


With respect to the proposed IEEE standards P754 and P854:
    The arithmetic diagnosed has unacceptable serious defects


- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Paranoia calculations elapsed time        4.4200 seconds
Paranoia report elapsed time              0.7100 seconds
```

N89-16363

Interfacing Ada* and Other Languages

Paul Baffes and Brian West

Intermetrics Inc.

INTRODUCTION

The Department Of Defence has mandated the use of Ada on upcoming projects involving embedded system software. NASA has also indicated that Ada has been baselined for the Space Station project. Both of these decisions will require the contractor community to transition from their current non-Ada programming environments. Existing software, that is proven and validated, will most likely continue to be used during the transition period. During this period new Ada programs and existing programs in other languages may need to be interfaced.

A myriad of possibilities exits for the solution of the transition problem. One set of solutions deals with translating the source code of the other language into Ada source code or the intermediate langauge used by the chosen Ada compiler. Another solution involves a special interface subroutine that switches from the Ada run time environment to the run time environment of the other language. The latter solution will be examined.

The above mentioned non-Ada programming environments consist of many different programming languages like FORTRAN, PASCAL and HAL/S. While each of these languages is unique, they are all members of the ALGOL family of programming languages and share many implementation characteristics. Therefore, an interface subroutine can be analyzed for any one of these language and the results can then be extended to the remaining languages. HAL/S was chosen for this examination.

The HAL/S 360 compiler which runs under the IBM MVS Operating System and the Ada compiler which runs under the IBM VM/SL Operating System were selected for this study. The primary criteria for the selection of the HAL/S and Ada compilers was that they were hosted on the same machine architecture. Both compilers were developed by Intermetrics.

GENERAL OVERVIEW OF THE PROBLEM

In this section, the general issues involved in interfacing any two different high-level languages will be explored. This explanation will outline the direction taken by the following sections, and should help in providing an overview of the intricacies involved in such an integration.

The Language Environment

Along with any Programming Language comes a set of assumptions under which that language is run. This set of assumptions can be called the environment of that language, and consists of both algorithms and data structures. While these may cover a variety of subject matter, most Algol-like language environments can be understood through a few basic ideas.

The first of these basic ideas is known as a run-time stack. In general, the run-time stack is used to keep track of local data and register contents across procedure calls as the program is being executed. In this way, the integrity of a procedure can be maintained while control is passed to a subprocedure, and then restored when the subprocedure returns.

Another of these basic ideas concerns the internal representation of data. For example, one language environment might use a signed magnitude representation for integers while another may use twos complement form. Naturally such details are not an interfacing concern when a calling procedure and the called subprocedure are written in the same language. However, care must be taken to ensure that these internal representations are identical when two different languages are involved.

A final basic concept involves the managing of run-time errors, commonly known as exception handling. Most often, a language environment will provide a large collection of procedures called a run-time

* Ada is a Registered Trademark of the U.S. Department of Defence (Ada Joint Program Office)

library which contains the mechanisms for dealing with these errors. However, since two different languages will use two separate run-time libraries, an error occurring in a subprocedure of one language would probably not be understood by the calling procedure of the other language. This would prevent the calling procedure from responding to the error in a proper manner.

The Basic Interface

In light of the previous discussion, the interface between two distinct languages becomes a matter of switching environments. To accomplish this at run-time, a special linking subroutine would need to be invoked from the run-time library of the calling procedure. This subroutine would provide the mechanism for saving the present environment of the calling procedure and initiating the new environment. In turn, upon termination of the called subprocedure, this subroutine would regain control and reinstate the old environment.

Ada provides an instrument for interfacing with other languages called the PRAGMA INTERFACE directive. The sections that follow take into consideration the details necessary for implementing this mechanism.

COMPARISON OF HAL/S AND ADA ENVIRONMENTS

Parameter Passing

Almost every subroutine makes use of parameter passing, whether it accepts some value or values as input, or produces some output, or both. This process of exchanging information between procedures is part of a language environment and thus will most likely vary from one language to another. In regards to the HAL/S and Ada compilers cited, the discrepancies are dramatic.

The Procedure Call

In most cases, all parameters are passed through registers to the called subroutine. However, when there are not enough registers for all of the parameters, another method is pursued. This method usually involves placing the parameter in temporary storage and passing the address of this location instead to the called subprocedure.

Both HAL/S and Ada comply with the conventions outlined above. However, the specific registers used by the two languages to accomplish these standards are not the same. For example HAL/S and Ada use a different register for addressing the temporary storage area where the overflow parameters are stored. In addition, Ada may store its parameters in more that one place, depending on whether or not they were dynamically allocated. Any interfacing subroutine would have to map one set of register conventions to the other and also be aware of the different locations where the overflow parameters are stored.

The Function Call

Functions, unlike procedures, return a value to the calling procedure. This value is returned via the use of a register. As was true with parameter passing, this register may contain either the actual value or a reference to the location where the value is saved. Again, each language will use different conventions for returning this value. In fact, the HAL/S and Ada compilers cited utilize different registers for this purpose.

Data Representation

A problem related to parameter passing arises from how each language chooses to represent its data types. There are a variety of factors involved in data representation including the number of bytes used, indexing schemes, value restrictions, and the algorithm employed for packing the representations to save space. Since each language will differ in its methods of representation, some scheme for converting data between representations would have to be implemented before any interfacing would be possible.

The Run-Time Stack

The objective of the run-time stack is to keep track of the flow of a program during its execution; namely, to record the dynamic nesting of the called procedures. To accomplish this, the run-time stack contains the information necessary to describe the state of the program at any point during its execution. The particulars of the run-time stack are also implementation dependent.

The HAL/S Run-Time Stack

HAL/S has a very straightforward approach to its run-time stack design. Its run-time stack is divided into "stack frames," one for each procedure currently being executed. These stack frames are further divided into two sections. The first of these is of a constant size and contains the following: a register save area, an area for the current code base, and a workspace for exception handling. The second section is of variable size and is used to store the procedure's local and temporary variables. The uses of these two sections are explained below.

When a subprocedure is called, a new stack frame is created and placed onto the stack. The contents of all the calling procedure's registers are then stored in the register save area of this new stack frame. In turn, when the called subprocedure returns control to the calling procedure these stored register contents are replaced into their appropriate registers. In this way, the calling procedure's register contents are not violated by the called subprocedure. The remaining fixed portion of the stack frame provides the procedure with run-time control information. This information includes: the location of the first executable instruction for the current procedure, a temporary workspace, and a link to the error library.

The second section of the run-time stack is left for the local and temporary variables of the subprocedure being executed. The size of this section varies from procedure to procedure depending on each procedure's number of local and temporary variables. The size of each procedure stack frame, however, is determined at compile time. So while stack frame sizes may vary from procedure to procedure, each procedure's particular stack frame size is fixed at execution time.

The Real Time Executive

Real time executives are used to synchronize and allow communication between two independently executing programs. Any program which depends upon some real world event will depend upon a real time executive for proper execution. The internal mechanisms which implement real time executives are nontrivial and vary widely among the languages that provide real time features. Although HAL/S and Ada both have a powerful set of real time executive tools, these tools are unalike and they require different approachs by the applications programmer for solving real time problems. Because their sets of real time executives are not the same, the HAL/S and Ada language environments will incorporate different implementation schemes. To interface these two sets of real time executives would pose an extremely involved challenge.

The Run-Time Library

Every language has a set of primitive utilities which it uses repetitively. This set of utilities is commonly called the run-time library. The run-time library is automatically linked with the program's object module before execution. As a result, every procedure or subprocedure of the program can employ any routine provided by the run-time library.

Of course, each language will have a unique run-time library. One of the more significant problems arising from this concerns error handling. When an error occurs during the execution of a program, the problem is most often managed by a routine in the run-time library. If this were to happen in a called subprocedure of a different language, there would be no guarantee that the process used to handle the error would be understood by the calling procedure. This problem is important because some errors may require termination of the program. Thus, if the called subprocedure were to force termination before returning control, the calling procedure would not be able to exit in a graceful manner. This could result in a loss of pertinent information. Additionally, similar errors may be handled with different levels of severity by different language environments. In particular, what may cause a HAL/S program to terminate may only raise an exception in an Ada program. This presents a formidable problem for the interfacing subroutine.

SUMMARY

Overview of an Interface Subroutine

The interface subroutine would operate in a straightforward manner. The routine would first load the passed parameters into the registers. A parameter would be passed either by its actual value or by a pointer, a machine address. Verifying that the parameters were passed in the correct format would be the responsibility of the Ada applications programmer.

The next step in the interface subroutine would be to initialize a new HAL/S stack frame and branch to the entry point of the HAL/S executable code. During execution, calls to the HAL/S run-time library may be made. To guarantee proper execution, the Ada applications programmer would have to include all needed HAL/S run-time library routines in the load module. Upon finishing the normal execution of the HAL/S code, a branch would be made back to the linking subroutine and the old stack frame would be popped off the stack.

Finally, the interface subroutine would remove the passed parameters from the registers. Before assigning these values to their appropriate memory locations, constraint checking should be performed. Any constraint violation should raise an exception and the corresponding exception handler should be invoked at that time.

Restrictions on the Interface

Restrictions, unfortunately, would have to be placed on the called HAL/S procedure. The interface subroutine would resolve as many of the differences between the two run time environments as possible. Those differences which could not be resolved would result in restrictions on the interface.

One restriction would involve the way

errors are handled. Run-time errors in the HAL/S executable code will not raise exceptions when they occur. Some of these exceptions could be raised by the interface subroutine when constraint checking is done. Other run-time errors in the HAL/S code would go unnoticed and the subsequent execution would be indeterminant. Note that the called HAL/S procedure would have to have an appropriate ON ERROR IGNORE statement or else the HAL/S code could make an unsupported operating system call.

Another restriction concerns the visibility of variables. At the point of the HAL/S procedure call in the Ada program, some of the declared variables may have visibility. While an Ada procedure called from the same point would be able to access these visible variables, the HAL/S procedure could not. Succintly, the only way the Ada program and the HAL/S procedure could communicate would be via the passed parameters.

Yet another restriction would be that the HAL/S procedure could not invoke real time executives. Additional restrictions may be to limit the use of Ada real time executives and to circumscribe the use of I/O in the HAL/S procedure. The above two proposed limitations need further investigation.

CONCLUSION

Interfacing two separately developed compilers is a complex task. The complexity arises because very few design standards exist for compiler development. This, coupled with the many complicated design decisions inherent in compiler construction, virtually guarantees noncompatibility. The interface subroutine which would link the two different run time environments would resolve as many of the dissimilarities as possible. The differences that could not be resolved would be responsible for the restrictions placed on the interface. Albeit restrictions would exist, the resulting interface may be well worthwhile.

BIBLIOGRAPHY

Aho, Alfred V. and Ullman, Jeffrey D., Principles of Compiler Design. Reading: Addison-Wesley Publishing Company, 1977.

Booch, Grady. Software Engineering with Ada. Menlo Park: The Benjamin/Cummings Publishing Co. Inc., 1983.

Ryer, Michael J. Programming in HAL/S. Cambridge: Intermetrics Inc., 1980.

Computer Program Development Specification for the Ada Integerated Environment. IR-MA-300. Cambridge: Intermetrics Inc., 1984.

HAL/S-360 Compiler System Specification. IR-60-07. Cambridge: Intermetrics Inc., 1981.

Deferred Binding in the Ada Software Support Environment

Paul Brown
Hikkok % High Technologies Lab
University of Houston-Clear Lake
Houston, Texas

This is a late paper. The author will provide copies at the conference.

N89-16364

SOFTWARE ISSUES INVOLVED IN CODE TRANSLATION OF C TO ADA PROGRAMS

Robert Hooi, Joseph Giarratano
University of Houston Clear Lake

ABSTRACT

It is often thought that translation of one programming language to another is a simple solution that can be used to extend the software life span or in re-hosting software to another environment.

This paper examines the possible problems, the advantages and the disadvantages of direct machine or human code translation versus that of re-design and re-write of the software. The translation of the expert system language called C Language Integrated Production System (CLIPS) which is written in C, to Ada, will be used as a case study of the problems that are encountered.

1 FUNDAMENTAL CONCEPTS

1.1 Introduction

CLIPS is a rule-based expert system language developed by the Artificial Intelligence (AI) section of the Johnson Space Center. The programming language C was used in the original implementation of CLIPS, while Ada is used as the new target language.

In re-hosting the original version of CLIPS from C to Ada, two approaches were attempted. The first approach was direct code translation, while the second was a complete re-write and re-design of the entire software.

1.2 Direct Code Translation As A Possible Approach

The work involved in the development of large software systems often represents huge amounts of time and expense. Monetary investments and time involved in the development make it extremely desirable to continue using these software systems for as long as possible. A few reasons for re-hosting to a new hardware or software environment are:

- o software system is still needed
- o difficulty in locating technical support
- o need to increase software versatility
- o greater execution speed
- o more economical hardware

At first sight, code translation may be seen as a simple, inexpensive approach to a complex and difficult problem. Translation seems to offer an attractive patch in extending the versatility and life span of existing software systems without the need to "re-invent the wheel".

1.3 Advantages Of Direct Code Translation

Direct code translation is often considered a very direct, simple and desirable method of re-hosting existing software to another environment. It offers a number of plausible advantages that can be highly questionable in retrospect. These advantages are:

- o elimination of some of the software life cycle phases
- o requires less time and effort compared to re-design and re-write
- o easily extended life span and software versatility
- o elimination of human errors compared to re-design and re-write.

These advantages will now be discussed in more detail.

1.3.1 Elimination Of Some Of The Software Life Cycle Phases

The major phases of the software life cycle [1] include:

1. requirement analysis	2. specifications/requirements
3. design	4. coding
5. verification and validation	6. maintenance and operation

If carried out correctly, the most difficult work involved in the software life cycle is in the early phases. Maintenance and operation may be time consuming but lacks the complexity of the first phases (1 - 3) of the life cycle, unless major changes are desired after the software is released. In fact, studies have shown that maintenance may account for up to 90% of costs for the software life span [8,9]. One of the major reasons for the development of Ada was to reduce maintenance costs. Direct code conversion offers a simple short cut to avoid the early phases (1 - 3) of the life cycle by:

- o requiring only source code of the software
- o minimizing verification and validation
- o allowing re-use of test data from the original
- o eliminating the need to do design conversion
- o eliminating the need to understand the functionality of the software, which is especially useful if the original programmers have left

In cases where the documents of the early phases of the software life cycle are missing or are poorly defined, direct code translation means eliminating "re-inventing the wheel". There is no need to derive a design if it is missing or to study and redefine poorly written documents.

1.3.2 Requires Less Time and Effort Compared to Re-Designing and Re-Writing

Direct code translation appears to be an attractive approach in that it is theoretically a simple, mechanical process if the host and target language are similar. There is little need to understand the actual requirements, design or functionality of the program. Much smaller machine and human resources are needed in performing the translation. All that is required is a basic understanding of the software tools and their interfaces, detailed knowledge of the host and target languages, and the different hardware specifications.

Once a translator is built, code translation may proceed very quickly and without the possibility of human error. In theory, translation work is basically tedious but mechanical and simple in nature. All that is required is a consistent, correct and accurate equating of the original implementation with that of the target language and its' environment. Also, once a translator is available, it may be used on other software or the translator may be sold for a profit.

1.3.3 Easily Extended Life Span And Software Versatility

A re-write and re-design of a software system is expensive and time consuming. It requires a considerable amount of professional human expertise compared to direct code translation which could be performed by either human or machine translators. If direct code translation is done by human translators, it may be expensive and time consuming, but it is still cheaper than a re-design and re-write. The early phases (1-3) of the software life cycle can still be skipped.

If the work is performed by a machine translator, it would still be relatively inexpensive since the only real work would be in the validation and verification of the accuracy of the results. A certain amount of editing and debugging may be required, but the work involved is relatively mechanical in nature while the resources needed are still less than an actual re-design and re-write.

The elimination of most of the work involved in the software life cycle, plus the possible availability of a machine translator and the ease involved in the work, could mean a saving in time. The re-hosting and re-targeting work can be completed in a relatively short period of time compared to re-design and re-writing.

1.4 Elimination of Human Error

Translation of computer software can be carried out either by a human or machine translator. If the software is large, then the use of a machine translator may be the least expensive approach, whereas for small programs, a human translator may be the better choice. The translated version is tested for accuracy and

correctness by computer programmers.
The requirements for a human translator is an understanding of the computer languages and external interfaces involved in the translation process. The work in general is very mechanical in nature. The advantage of the mechanical translator compared to a human is a reduction in software errors in the translation. The disadvantages of machine translators is that the human may clean up the code during translation because the human can understand the semantics as well as just the syntax.

2 SOFTWARE ENGINEERING ISSUES INVOLVED IN DIRECT CODE TRANSLATION

2.1 Introduction

The process that direct code translation generally takes often results in a failure to address certain design and implementation issues in software engineering. These can develop into major problems later on in the software life cycle. They are:

o differences and incompatibilities in design methodologies

o differences and incompatibilities in language implementation

o possible disregard of the richness of the target language

o possible inaccuracies and discrepancies between languages

o maintenance costs may well exceed savings of translation.

Unless the above issues are addressed, the problems and disadvantages may outweigh all the advantages made in a direct code translation.

2.2 Differences And Incompatibilities In Design Methodologies

The types of available software tools have a profound effect on our thinking process and thus the design and development of the software. The types of design methodologies used are often guided as well as restricted by the software tools used in the implementation of the actual program. It should be noted, however, that methodologies are generally much easier to compromise than the implementation language tools. The same rules apply to modern software engineering principles and practice.

The principles of modern software engineering as incorporated by languages such as Ada and Modula-2 are [2,3] are:

o modularity	o abstraction	o information hiding
o localization	o uniformity	o completeness
o confirmability		

If the programming language used does not directly provide support towards the above software engineering principles, then it is difficult and often impractical to implement a design which adheres to these concepts. The implementation language and design methodologies used should be mutually compatible for best results.

So converting a BASIC program to FORTRAN IV would be reasonable since they share similar software principles. Likewise conversion from Modula-2 to Ada would be feasible since both languages support the above modern engineering principles. Difficulties arise in the translation of programs in a language like C to Ada since C does not adhere to the above principles of Ada.

The use of an object-oriented design methodology [4], together with an object oriented language such as Ada, forms a very highly compatible choice towards the support and implementation of these software engineering principles [10]. If a programming language does not readily support these concepts it will probably be absent in the implementation. In languages which do not have this support it may be too expensive and difficult to enforce these principles. In most cases, the designers and implementors would probably choose a design methodology that allows an easier implementation, rather than one in which the language would have difficulty adhering to.

A major issue involved in directly translating a program to a target language is that the type of methodology used is often ignored. If the work is performed by computer programmers, then it may be possible to modify and adapt some of the code to that of Ada's object-oriented approach. It would be impossible for a simple syntax-directed machine translator to do this completely, since it involves a certain degree of independent thinking, analysis and understanding of the original software. Thus, a machine translator would have to understand the semantics as well as the syntax to do a thorough job. Such a translator would have to include artificial intelligence and expert system techniques and would be very difficult to build. A simpler alternative would be to have a human examine the code produced by the simple translator and polish it up. However, this could still be a major task.

If the original implementation is not an object-oriented design methodology, then it will not normally be present in the translated version. For example, if the original does not support the concept of information hiding, then the translated version will not. If the original design methodology adheres to the concept of data flow decomposition or the Jackson Design methodology [5], then the translated version certainly would not have any of Ada's object-oriented approach.

A time factor should also be taken into account since the type of methodology used is dependent on when it was first conceived. Ada's object-oriented design methodology would certainly be absent if the software was developed prior to the 1980's. This technical gap may not be easily bridged in direct code translation unless the languages are similar, such as Modula-2 and Ada.

The ability of the language to support these methodologies must also be considered. For example, Ada's packages supports the concept of information hiding, which may be simulated by C's statement "INCLUDE". However, this does not mean that C provides the same capabilities or support of the concept of information hiding found in Ada. There is no close equivalent in C to Ada's private and limited private types or visibility controls.

Translation becomes even more difficult concerning the concept of re-usable software components. For example, there are no facilities in C to directly simulate Ada's generics.

A major difficulty in translation occurs when the documentation is missing. The problem is compounded when the methodology used is unknown and is not similar to Ada's object oriented approach. These problems were found in the translation of CLIPS.

A certain amount of re-design and re-write was required in certain program segmnents in order to conform to the language implementation requirements of the target language (ADA). An example is the difference between a C library program versus Ada's packages. Each C library program has the function - "main", which may make calls to other external library functions or functions within the same file. The visibility rules in C allow calls by the sub-program unit "main" to other functions located anywhere within the file dependent upon the programmer's convenience. Ada's visibility rules allow procedures and functions to be called by other program units only if declared above it. An example found in the CLIPS demonstrating C's visibility problems is shown below:

```
main ()
   {
          :
      command_loop ();
      if (opt_u_found == TRUE)
         {displayfunctions();}
     }
       :
command_loop ()
       :
displayfunctions ()
```

Ada's visibility rules would require:

```
procedure Command_Loop is -- assumes converted to a procedure --
     :
end Command_Loop;

procedure Display_Functions is
     :
end Display_Functions;

procedure Main is
   :
begin
   :
   Command_Loop;
   if ( Opt_U_Found = True ) then
      Displayfunctions;
      :
end Main;
```

In view of the differences in design methodologies, it follows that if the translation does not include the methodologies, then the work is only partially complete. A translation without the design methodology is not a true representation of the target language's environment. It is therefore not possible to re-target software correctly by direct translation if the design methodologies are not considered in the work.

2.3 Differences And Incompatibilities In Language Implementations

Discrepancies and incompatibilities between different computer languages mean that what is considered as an acceptable programming practice in one may not be permitted in another. C has weak typing, which means that unless it well enforced, most data types can take on any values assigned to them. If the program is to be properly translated to Ada, then a number of conversions and data checks must be included to restrict the values assigned to variables. This is needed to accomodate the differences between C's weak typing versus Ada's strict typing requirements.

The strength in Ada's requirement for strict typing enforces program reliability and consistency, while C allows for greater flexibility on the part of the programmer. The result in accomodating the typing requirements of Ada is that the translated version is seldom, if ever, smaller than the original. In the translation of CLIPS to Ada, it was found that for every line of C code, the average is generally two lines of Ada code. This does not mean that Ada is a less efficient language compared to C, merely, that Ada's strict typing enforces consistency and provides a more reliable program. This is particularly important to the Space Station since much of the software will support human lives and also directly affect the longevity of the space station.

An example from CLIPS demonstrating C's weak typing which must be corrected in the translated version is shown below:

```
    :
float tally = 0;
    :
char lm;
    :
int ten = 10;
    :
tally = tally*10 + (lm - 'o');
```

The Ada version must have the following changes made:

- o convert integer 10 to float
- o convert data types: lm and 'o' to ascii values
- o convert the resulting arithmetic operations (lm - 'o') to float
- o value initialized to tally changed to 0.0

```
tally := tally*10.0 +
   float(Character'Pos(lm) - Character'Pos('o'));
```

The complexity of the problem increases if the typing problem occurs in the arguments of a subprogram call. Data conversion will have to be made prior to actual passing of the values to the subprogram call.

In addition to the weak typing problem, certain language features in C which are not found in Ada have to be worked around. This again accounts for some extra code being produced. An example from CLIPS showing the auto increment is:

```
while ((atemp != null) &&  (++count != nnn))

           versus Ada's version

while ((atemp /= null) and (count /= nnn)) loop
   count := count + 1;
          :
end loop;
```

Note that in this case, the lack of an auto increment or decrement in Ada does not necessarily mean it is a slower language at run time. Depending upon the compiler implementation, the functionality is the same and should execute at similar speeds. The major difference is that Ada aids readability, thus making it easier to understand and maintain.

The extra code size may present several important problems:

- o program efficiency could be sacrificed
- o storage and execution speed becomes worse
- o maintenance problem increase due to increased code size

Depending on where the increased code is generated source code, code size could result in slower program execution. In a situation where response time is crucial, such as real time execution, anything that may reduce execution speed should be examined very carefully to see if it could be acceptable.

For software systems that are relatively small, an increase in size may not pose an important issue. However, as the magnitude and complexity of the software increases, there will be a proportional hardware demand. For example, consider a large embedded software program occupying 100,000 blocks of disk space. Increasing the code size by two times might exceed the remaining disk capacity. If this rule is applied to software systems that are even larger, then size requirements made by direct code translation may not be an acceptable solution.

An increase in code size would also mean that the complexity of software maintenance would also grow. Issues in software maintenance will be further examined later in this paper.

Some of the results found in the translation of CLIPS to Ada:

Comparison of storage size for one of the files on the VAX:

original version:	CLIPS.C	occupies 175 blocks
translated version:	CLIPS.ADA	occupies 369 blocks

Comparison of code size for functions:

Excluding global data declarations, for function Rarray the original occupies approximately 15 statements
Excluding global data declarations.
translated version occupies 26 statements.

Some factors contributing to an increase in code and storage size are:

- o statement terminators found in Ada - end if, end case and others
- o instantiations of generic I/O packages
- o path names used in calls made to other packages
- o absence of auto increment and decrement statements
- o absence of statements with embedded functions and statements in boolean tests, such as auto increments

Additional explanation of the reasons for the increase in code and storage size will be discussed in the next section.

2.3.1 Possible Disregard Of The Richness Of The Target Language

In order to translate as accurately as possible, the simple syntax method is to equate statements found in the original with that of the target language. This presents a disadvantage in that much of the richness found in the target language is often ignored. If the translation is done manually, then certain segments of the original could be re-built to allow better usage of the target language. The same cannot be easily applied if the work is performed by machine translators unless semantic understanding is also included.

Ada has a standard of 63 reserved words regardless of the implementation versus C's approximate 33 (including functions for the C preprocessor). These 33 words of C depend upon the compiler, version and host environment. Ada has, in addition, a number of features which are not present in the standard C implementation. They are:

- o predefined language attributes
- o predefined language pragmas
- o predefined language environment:
  - o language predefined identifiers (package standard)
  - o utility packages such as system and calendar
  - o input and output packages
- o ability for overloading, generics, multi-tasking, nested generics and packages

The use of generics would drastically reduce the amount of code found in the original, since functions with like actions but different data types and properties can be grouped together and placed in the same subprogram. As a generic unit, a template is built to accommodate the function of a sub-program without specific properties. The instantiation allows the properties to be set to the generic package.

While C allows for greater flexibility in usage, the richness of Ada permits better control, reliability and flexibility in the programming environment. For example, in order to recover from run-time errors, the C program will have to simulate what Ada naturally does in its ability to raise and handle exceptions. A direct syntax translation would result in having a simulation of run-time error recovery in Ada, which ignores what the language is equiped to perform naturally.

An example taken from CLIPS is:

```
if (notstate == 0)
   {
    if (btemp == NULL)
      {
       htemp -> locals = valuescopy(line->locals);
```

If the power of Ada is exploited correctly, then the structure above could be combined, yet simplified as follows:

```
 if ( Notstate = 0 ) and then ( Btemp = null ) then
    Htemp.Locals := Valuescopy ( Line.Locals );
```

It should also be noted that since Ada data types are not case sensitive, then capitalization of the variables could be used to improve readability, and so provide better maintainability.

The full power of the target language is seldom exploited completely in direct translation. The increase in code and storage size of the translated version is in no way an indication that the original host language is a better software tool. The same rule applies if the execution speed of the target language is reduced. It does not mean that Ada is a less efficient language, merely that it is not exploited fully.

2.3.2 Possible Inaccuracies And Discrepancies Between Languages

Translation of language syntax is generally a very mechanical process. To equate accurately, it necessary to consider the semantics of a program, which is a much more difficult task.

The difficulty of the problem of correct semantic translation increases with the magnitude and complexity of the software. In addition, if the source in the original is poorly written and has a very confusing implementation, the chances of a misinterpretation increases. The main software issue is the program's reliability, accuracy and correctness. If the semantics are misconstrued in a subtle area that is difficult to detect, then locating and debugging the logic problem would be equally difficult.

The differences and restrictions in language implementation are a major cause of discrepancies in translation. For example, C permits recursion for the arguments in a function call since the values passed into the function can be changed. In contrast, the parameters in Ada must be of a formal type, and changes to those values are not allowed. To work around this problem, the translator must decide whether to declare the values that are changed in the function as global data types or convert it into a procedure. If the values are changed into global data types, then the issues of localization and modularity are raised. In addition, care must also be taken to ensure that those global values are correctly initialized, changed or kept at each call. If the values are not traced correctly, then the program execution may not function as originally designed or there will be a set of global values created at every subprogram unit that makes a call to that function. If a function is converted into a procedure, then the calling process made by the subprograms will have to be changed.

An example from the CLIPS demonstrating the changes made to the arguments in a function as follows:

```
Any(code, values)
int code;
    :
     if (values->whoset == code)
         ret = -1;
     else
        values = values->next;
    :
    return(ret);
```

Note that in C the arguments of a function are value parameters. It can, however, perform as a variable, formal or value parameter. Ada strongly enforces the type of parameter used, which is defined in the subprogram arguments. For example:

```
push ( first, second, third )
```

versus Ada's parameters

```
procedure Push
   ( First  : in Integer;
     Second : in out Float;
     Third  : out    Boolean );
```

This ensures program reliability and consistency, as values passed in are restricted to performing within the scope of their declared type. The simple solution in translating from C to Ada is to have all arguments declared as value parameters. In translating the C code to Ada, unless checks are made to determine if the arguments passed perform as a variable, formal or value parameter, this particular strength in Ada will be ignored.

Another possible semantic problem in direct translation is that C is a case-sensitive language. A data type with the same name but written in upper-case is a different variable to that which is in lower-case. Caution must be taken to ensure that data types with the same names but different cases be given different names. In addition, variables in C may be reserved words in Ada. The translator must be able to identify these and assign meaningful substitutes.

Examples taken from CLIPS to the problem above is shown below:

```
struct element *out;     extern struct internode *AGENDA;
       :                                  :
while (out != NULL)     struct internode *agenda,*step,*past;
       :                                  :
```

```
    out = out->next;                    AGENDA = agenda;

          (1)                                 (2)
```

In example (1) a compilation error would result if the translation process does not substitute a different name to the data type - "out". The data name "out" is an Ada reserved word and cannot be used as a variable name.

Example (2) can have unpredictable results, depending on the translated version. Since Ada is not a case sensitive language, the translated statement could really be doing nothing, unless a change is made to either one of the two object names - "AGENDA" or "agenda".

The ability of C to include function calls in test statements further complicates the translation process since a patch must be used to adapt to Ada's language requirements. Temporary variables must be used in order to obtain the values required for the boolean tests prior to the execution of the statements. Again, the issue is not that Ada is a less efficient language, but that it enforces program readability for better maintainability. An example from CLIPS shows the problem:

```
    if ((any(go,list) == -1) && ((second == -1) ||
       (any(second,list) == -1)))
```

while an Ada patch solution would be:

```
            :
  First_Value, Second_Value : Integer := 0;
            :
  First_Value  := Any ( Go, List );
  Second_Value := Any ( Second, List );
            :
  if ((First_Value = -1) and
     (Second = -1 ) or
    (( Second_Value = -1))) then
```

Bit manipulation [6,7] is another area that Ada does not <u>directly</u> support, but is present in C. The translator must be able to use an Ada implementation of the compiler that can perform a representation of the size of the bit used. There are also bit manipulation operators in Ada similar in C. A patch must be found in order to translate correctly and accurately. Note that this problem did not arise in the translation of CLIPS as there was no bit manipulation used.

In certain cases it may not even be possible to implement a direct translation. For example, problems arise when the original implementation performs systems calls using operating system dependent control languages such as IBM JCL,DEC BLISS and DCL. Problems occur also when the target language does not contain the necessary interface features. Direct code translation is thus dependent upon the implementation capabilities of the target language and its host environment.

<u>2.4 Maintenance Costs May Well Exceed Savings Made in Translation</u>

The quality of the simple syntax translation is at best equivalent to the original. In most cases it is inferior to that of the original. The reason is that direct translation copies over the raw design and implementation of the original. If the source code in the original is unstructured, cryptic and consists of meaningless data names, then the translated version would bear the same resemblance. As the old saying goes, "Garbage in, Garbage out". Just because a program is translated to Ada does not automatically make it a good program. In addition, the increase in code to patch some of C's weak typing serves only to complicate the task of maintaining the software.

In any language, if the original has poorly designed code, it will be hard to translate or re-design correctly. Ideally, the person who wrote the original code should also have been trained in the target language. For example, the best person to write C code would be an Ada programmer who knows C. A programmer who has experiences with a more evolved language such as Ada, will write better C than one who knows just C.

An example an equivalent translation from CLIPS is:

```
if (((element->state == 'o') || (element->state == 'n')) &&
   (element->type != FCALL) && (element->type != COAMP ))
    {
  while ((list != NULL) && (stop == 1 ) && (go == 1))
      {
    if (element->name == list->name)
       {
       go = 0;
       pkg = 0;
       if (element->type == NUMBER)
          if (element->ivalue != list->ivalue)
             stop = -1;
               :
```

The translated version in Ada is shown following:

```
if (((element.state = 'o') or (element.state='n')) and
    (element.type /= FCALL) and (element.type /= COAMP )) then
        :
 while ((list /= null) and (stop = 1) and (go = 1)) loop
    if (element.name = list.name) then

       go := 0;
       pkg := 0;
       if (element.type = NUMBER) then
          if (element.ivalue /= list.ivalue) then
             stop = -1;
               :
```

Note that the code and structure characteristics present in the original can also be found in the translated version. These are:

o meaningless, cryptic object names

o lack of capitalization standards for readerability o poor structures and language usage

Code from CLIPS that has data types with meaningless names like jill, jack, junk, grab and has no documentation, will produce a translated version with the same characteristics.

In the simple syntax translation of CLIPS to Ada, the general results obtained were that the translated version was worse than the original. It is at best an Ada program written in C methodology, with Ada structures looking like C structures. A simple syntax translation of bad C code will produce bad Ada code. However, this does not mean that good C code will produce good Ada code, since much of the wealth of Ada is ignored. This defeats the purpose of the translation to Ada, which is supposed to improve maintainability and reliability. If the software lacks quality, it cannot be easily built on, understood, modified and most important - maintained.

2.5 Summary

In view of today's rising software costs, where the bulk (80% - 90%) of the expense lies in maintenance and operation, direct translation may not be the best alternative in extending the versatility and life span of a software system. It is at best a patch and at worst an expensive solution when maintenance is considered.

3 DIRECT TRANSLATION VERSUS RE-WRITE AND RE-DESIGN OF COMPUTER SOFTWARE

3.1 An Evaluation Of Direct Translation

Simple syntax code translation may not be the ideal solution to a difficult and complex problem. Yet it is not a totally useless approach since there are certain values that are tied to the process. For example, if the program is relatively small, simple, and has a limited life span and usage, then translation may well be the best approach. In certain cases, where a design document is not present, a translation may be a possible method used to build a prototype for study purposes prior to the actual re-write of the entire software.

Because of the expense involved, human translation is generally limited to small and simple programs. The cost of human re-writing and re-design is best served in real-time code where performance is critical. In the translation of CLIPS, it was found that the time spent in the translation process was almost equivalent to that used in the original implementation and was thus self-defeating.

Simple syntax translation should be avoided if the target environment has a very different design methodology. It can be strongly considered if the target is a different host machine or a new version of the same language and design methodology.

3.2 An Evaluation Of Re-Design and Re-Write

The re-design and re-write approach should be strongly considered if a well documented design along with the specification exists. The original types of methodology used may be incompatible with the target , but it may be converted and adapted to the requirements of the new design methodology. The reason is that if the design is clear and well documented, it can be easily understood, worked upon and modified to fit any methodology. Coding is a relatively simple and mechanical process if a good design exists. The most difficult work involved in the development of any software is still the early phases of the life cycle. If a design exists, it can be studied, the weaknesses can be avoided in the implementation, and the strengths enhanced further.

The only part of software that can be transparent to all languages and host machines is the design and its' specifications. Once the design is converted to suit the requirements of the new methodology, it can be ported to the new target language and host machine. "Re-inventing the wheel" can be avoided only if a design is present.

3.3 Summary

Simple syntax translation or re-design and re-write are alternatives that can be used, but these have to be carefully considered before either one is adopted. Re-design and re-write should be strongly considered if a design is present. Translation may be considered if the goal is to port the software to a different host machine or up-date the software.

Considering the fact that neither of the two approaches is exactly easy to adopt, a few possible alternatives can be taken into account. These are:

- o interface the original with Ada
- o implementing the new software in Ada and port the data produced by the old software to be processed in the new environment

Interfacing Ada with other languages can be done using the Ada language's predefined pragma INTERFACE. Consideration should be given to the possible restrictions due to the different implementations of the compile. The reason being that this pragma is an implementation feature dependent upon the Ada environment. Certain implementations may allow for full usage, while others may be used partially and some none at all [6].

In cases where the host hardware is out-dated and an Ada compiler may not be available or an interface with the target language cannot be made, then it may be advisable to use the old software to generate the data. Any additional processing that is not dependent on the old software may have the new implementation developed in Ada. The data generated can be ported and executed in a new host environment.

Conclusion

Direct code translation or re-write and re-design may not be the only available solutions. There are basically no cheap and easy solutions to the problem. In terms of today's need to reduce the cost of software maintenance, plus the greater importance of software reliability, it may be much better to rebuild the entire system correctly. The advantage is that the faults and weaknesses are known and can be avoided. A better, more reliable software system can be built in place of the original.

Acknowledgements

Support for this work was provided by NASA Contract NAS9-17010, Task No. B13.C

References

1. Giarratano, Joseph, Foundations of Computer Technology, pub. by Howard W. Sams, 1982.

2. Douglas T. Ross, Hohn B. Goodenough, C.A. Irvine, "Software Engineering: Process, Principle, and Goals" Computer, May 1975.

3. B. W. Boehm, J. R. Brown, M. Lipow "Quantitative Evaluation of Software Quality", Proceedings of the Second International Conference on Software Engineering, pp. 592-605, 1976.

4. Grady Booch, "Object Oriented Development" IEEE Transactions on Software Engineering, Vol. SE-12, No. 2 February 1986.

5. J. R. Cameron, "Two Pairs of Examples in the Jackson Approach To System Development" Proceedings of the 15th Hawaii International Conference on System Sciences, January 1982.

6. Donald G. Martin, "Non-Ada to Ada Conversion", Journal of Pascal, Ada, & Modula-2, Vol.4, No.6, pp. 36-40 (1985).

7. Douglas L. Brown, From Pascal To C - An Introduction to the C programming Language, Wadsworth Publishing Company, 1985.

8. Girish Parikh, The Guide to Software Maintenance Winthrop Publishers, Inc, 1982.

9. Ian Sommerville, Software Engineering Addison-Wesley Publishing Company, 1985.

10. Mark W. Borger, "Ada Software Design Issues" Journal of Pascal, Ada, & Modula-2, Vol. 4, No.2, pp. 7-14, 1985

N89-16365

REHOSTING AND RETARGETING AN Ada COMPILER

A DESIGN STUDY

Ray Robinson

Harris Government Systems Sector, Software Operation

The goal of this study was to develop a plan for rehosting and retargeting the Air Force Armaments Laboratory/Florida State University Ada cross-compiler. This compiler was validated in September, 1985 using ACVC 1.6, is written in Pascal, is hosted on a CDC Cyber 170, and is targeted to (i.e., produces code for) an embedded Zilog Z8002. The study was performed to determine the feasibility, cost, time, and tasks required to retarget the compiler to a DEC VAX 11/78x and rehost it to an embedded U.S. Navy AN/UYK-44 computer.

Major tasks identified during this study were rehosting the compiler front-end, rewriting the back-end (code generator), translating the run-time environment from Z8002 assembly language to AN/UYK-44 assembly language, and developing a library manager. The Navy's MTASS/M software development tool set, which is currently used to develop FORTRAN, CMS-II, and assembly programs for the AN/UYK series, will provide an assembler, a linker-loader, and a simulator. Reuse of these three tools reduces the project's cost by at least a factor of two and eliminates the risk associated with developing them.

The resources required to rehost and retarget the compiler were estimated at nine people for fifteen months, for a total of 135 months of effort. The product of this effort will be an Ada cross-compiler which passes the ACVC 1.6 validation tests. The unpredictable cost of validating the compiler using the latest version of ACVC is not included in this estimate.

The author was the technical leader of the rehost/retarget study and is currently involved in the development of the Tester Independent Software Support System (TISSS). His education includes a B.S. in Engineering from U.T. Chattanooga, a M.S. in Chemical Engineering and a M.S. in Information and Computer Science from Georgia Institute of Technology, and work in progress leading to a M.S. in Space Technology from Florida Institute of Technology.

Ray Robinson
505 John Rodes Blvd., Bldg. 1
Melbourne, FL 32902
(305) 242-5678

Considerations for the Task Management Function of the NASA Space Station Flight Elements' Operating System Software

Larry Fishtahler
Computer Sciences Corporation
Silver Spring, Maryland 20910

This is a late paper. The author will provide copies at the conference.

N89-16366

The TAVERNS Emulator: An Ada simulation of the Space Station Data Communications Network and software development environment

by Dr. Norman R. Howes

Introduction.

The Space Station DMS (Data Management System) is the onboard component of the Space Station Information System (SSIS) that includes the computers, networks and software that support the the various core and payload subsystems of the Space Station. Although some of the DMS software runs in the subsystem computers, the subsystem computers themselves are not considered to be part of the DMS. Also, the applications software that is specific to a subsystem (e.g., the Communications and Tracking Subsystem) is not considered part of the DMS.

The various core subsystems (there are 22 of them) are to be implemented on Standard Data Processors (SDPs). This does not imply a standard computer has already been selected for this role but that all subsystem computers are to have the same instruction set architecture (ISA). It is also possible that a single SDP may host more than one subsystem. A diagram of the DMS together with the various subsystems is shown in Figure 1.

Figure 1 shows each SDP connected to a Core or Payload network via a Network Interface Unit (NIU). The NIU is itself a computer, probably with the same ISA as the SDP. The NIU hosts the Network Operating System (NOS) component of the DMS. On the other hand, the SDP hosts the application software for one or more subsystems. The SDP has an operating system (OS) of its own that is some times referred to as the local operating system (LOS).

A great deal of the DMS software resides in the SDPs. Those parts of the DMS software that provide the file management capability and the data base management capability are examples of DMS software that resides in the SDPs. Most of the DMS software that supports the actual transmission of data (both datagrams and virtual circuit transmissions) resides in the NIU and is referred to in general as the NOS.

The Ada packages of services available to the core or payload application programmer for (1) network communication, (2) file management, (3) database management, (4) data acquisition and distribution and (5) crew workstation services are documented in the DMS Test Bed Users' Manual (NASA/JSC No. 22161).

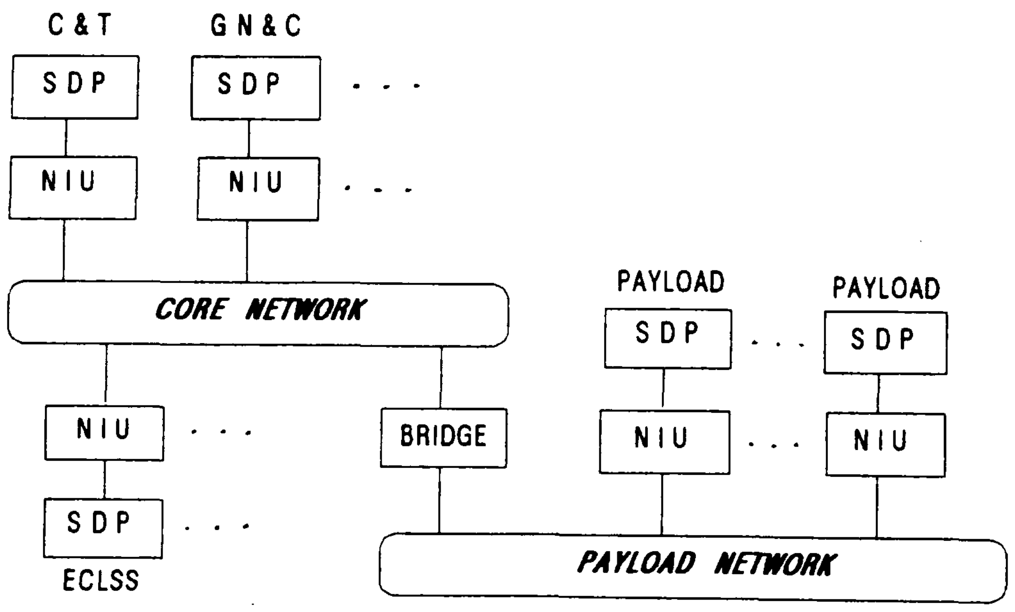


Figure 1

The TAVERNS Concept.

TAVERNS is a distributed approach for development and validation of application software for Space Station. The acronymn TAVERNS stands for Test And Validation Environment for Remote Networked Systems. The TAVERNS concept assumes that the different subsystems will be developed by different contractors who may be geographically separated.

In this approach, each software development contractor for the station will be provided with a miniature version of the Space Station DMS complete with three SDPs. One of the SDPs is for developing the subsystem software, one hosts the Displays and Controls software and the third hosts a simulation of the network core subsystems (e.g., ECLSS, C&T, GN&C, etc.) and the network loads. A diagram of such a TAVERNS DMS Emulator is illustrated in Figure 2.

The SDPs and NIUs on this mini Space Station DMS will host the same Services as the real DMS, so to the applications programmer, it will appear that the entire Space Station DMS environment is present. In turn, these TAVERNS systems will be interfaced with the Space Station SSE (Software Support Environment) and there will also be a TAVERNS on the station. In this way, software can be developed and checked out by different contractors at different locations. Completed and tested applications can then be transferred to the SSE for validation.

After an Initial Operational Capability (IOC) has been achieved for the Station, new validated software modules can be transmitted to the station where they will be revalidated on the onboard TAVERNS before being placed in service.

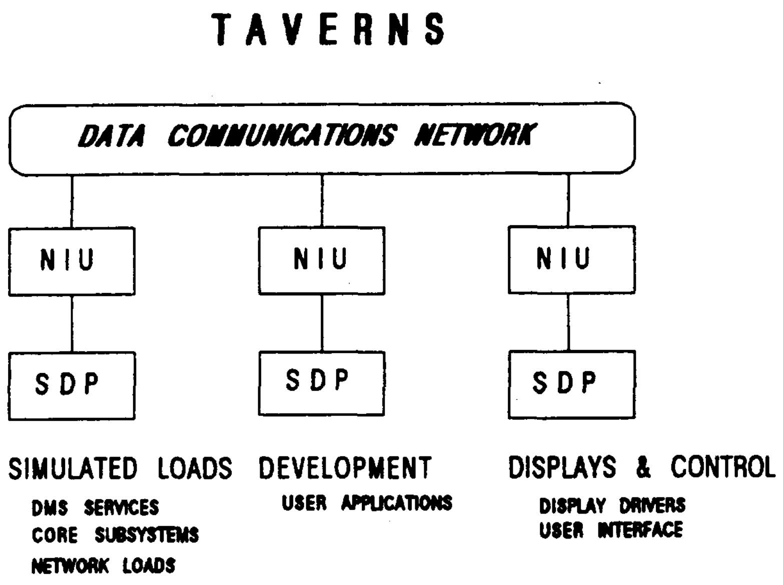


Figure 2

The TAVERNS Emulator is an Ada simulation of a TAVERNS on the ASD VAX in Building 16A. The software services described in the DMS Test Bed Users' Manual are being emulated on the VAX together with simulations of some of the core subsystems and a simulation of the DCN. The TAVERNS Emulator will be accessible remotely from any VAX that can communicate with the ASD VAX.

The purpose of this simulation is to (1) test the functionality of the DMS Services as documented in the DMS Test Bed Users' Manual, (2) provide a DMS software environment that is consistent with the one described in the Users' Manual where subsystem test bed developers can attempt to interface their subsystems with one another and (3) provide an environment where the TAVERNS concept itself can be evaluated and improved.

Ada features of the TAVERNS Emulator

Purposes (1) and (2) above are of special interest to the software engineer or programmer who will be designing or coding programs in Ada. In a way, the DMS Services as described in the Users' Manual can be thought of as an extension of the Ada language for distributed applications. These services are actually packages of utilities (subprograms or tasks) for performing certain operations such as transmitting a message from one subsystem to another or opening a file at a remote node (subsystem) and reading records from it.

These utilities are not only written in Ada but their intent is to operate on Ada data structures in a transparent manner. For instance most data communications networks with which anyone has any experience only allow a user to transmit data in a single predefined format such as in ASCII or binary packets. These packets may be as small as a single character as with asynchronous communications or a large binary block as with synchronous communications. On the other hand, when using an object oriented language like Ada what one would like to do is transmit an entire Ada object without first having to convert it into an ASCII or binary string.

The DMS Test Bed Users' Manual describes Ada oriented utilities such as this for transmitting Ada objects, writing Ada objects to remote files, etc. In fact, the Users' Manual describes even higher level services that use various Ada objects. For instance, one of the most frequently needed communications capabilities for the Space Station core subsystems is a request to read a set of measurements. By a measurement is meant the reading of a certain sensor (such as temperature or pressure) or the determining of the state of something (such as a valve being open or closed).

For most applications more than one measurement needs to be read at a time. The DMS provides a service for assigning logical "set names" to a set of measurements and a service for requesting the reading of a whole set of measurements by issuing a single command. When such a request is made, the DMS returns all of the readings in an Ada structure that depends on variant records that is optimized for this application and is independent of the Ada types that correspond to the various measurements in the measurement set. Furthermore, not all the measurements in the set have to be located at the same node on the network.

The Ada Simulation

The Ada simulation is being designed to run in a single VAX with access from another VAX. The intent here is for the VAX in which the simulation runs to represent all of the TAVERNS system except the SDP node and the other VAX represent the SDP. The user of the TAVERNS Emulator can develop Ada code on any VAX and then link to the TAVERNS Emulator VAX via the simulated DMS Services thereby simulating the way a contractor would develop Space Station applications software on the SDP node of a TAVERNS system. A diagram of the Ada simulation is shown in Figure 3.

The Ada simulation is being developed in two phases. The first phase configuration is shown in Figure 3. The first phase consists of a demonstration in which the user's only participation is that of responding to prompts on the display. In this configuration, the display handler software consists of the Ada procedures that control the menus from which the user chooses options during the demonstration and the various screens the user sees as part of the demonstration.

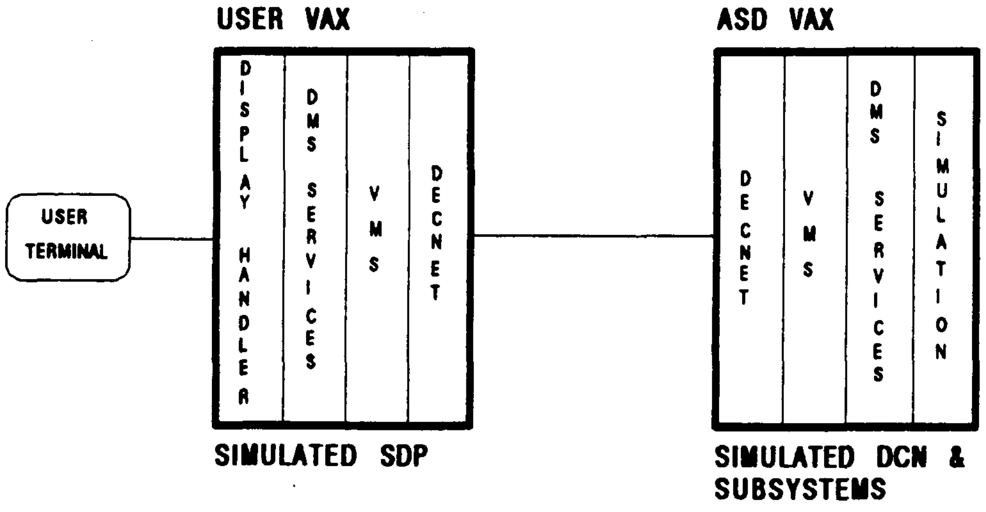


Figure 3

For the most part the DMS Services software maps the various DMS service commands onto the appropriate VAX VMS or DECNET service or combination of services to accomplish the specified DMS service. Where no existing combination of VAX services will accomplish a DMS service the necessary Ada subprograms are being developed.

The simulation software that runs in the ASD VAX is based on deterministic models of three subsystems for the first phase demonstration. These are the Communications and Tracking (C&T) Subsystem, the Environmental Control and Life Support (ECLS) Subsystem and the Mass Memory Management (MMM) Subsystem. The C&T and ECLS subsystems are modeled as a set of measurements. During the demonstration the values of the measurements change in accordance with a predetermined algorithm. The MMM subsystem is modeled as a set of data structures that relate which files belong to which subsystems and how files are related to each other through directories.

The demonstration consists of prompting the users for which sets of measurements the user wants to see displayed; using the DMS Services to request a reading of these measurements across the network (across the physical network between VAXs), to reply to the request at the other end of the network and to build the display in response to the user's request; and using the DMS Services to handle the supporting file management functions (the globally known measurement names are stored on a remote file). This last feature may seem a bit contrived as it was incorporated into the demonstration in order to insure that the functionality of the distributed file handling services of the DMS were tested. In the real DMS the globally known names may well be stored at every node.

The second phase of the simulation will consist of a set of transportable Ada packages for the user's host VAX that will enable the user to call the DMS Service utilities from user written application programs. The user's requests for remote services will be transmitted to the Ada simulation running in the ASD VAX for servicing. In the second phase simulation, the user's host VAX will appear as a node on the DCN and requests for local services will be considered to be remote since the simulation will only reside in the ASD VAX and not a remote user VAX.

Datagram Service Simulation

The Datagram Service is simulated on the VAX using six Ada tasks as shown in Figure 4. Three of these tasks run in each of the two VAXs involved in the simulation. Task SEM is a semaphore that controls access to the underlying DECNET network "file" (DECNET looks like a file to an Ada subprogram or task). Tasks INQUEUE and OUTQUEUE continually pass a token back and forth across DECNET until one of the OUTQUEUE tasks has a datagram to transmit. When this task gains possession of the token it transmits its datagram(s) and then goes back to circulating the token.

The simulated Datagram Services such as SEND or RETRIEVE are procedures that either place a datagram in an outgoing queue or fetch one from an incoming queue. The datagram service supported by the simulated DMS services is a very Ada oriented service in that the datagrams themselves are Ada objects that are prefaced by a header that contains the transmission parameters.

Package DATAGRAM is a generic package that a user of the Datagram Services instantiates for each different Ada object to be transmitted. At the receiving end, the type of object being transmitted can be determined by first examining the header.

The format of the SEND command for datagrams is:

SEND(MESSAGE,ADDRESS); or SEND(MESSAGE,NAME);

where MESSAGE is the datagram to be transmitted and ADDRESS is the logical network address of its destination. When the alternate form of the command is used, the parameter NAME is the name of a list of addresses to which the datagram is to be sent. The simulated Datagram Service supports a multicast capability for selectively sending datagrams to a list of predefined addresses. The command for assigning a logical name to a list of addresses is the MULTICAST command and its format is:

MULTICAST(NAME,AddressList);

where NAME is the name to assign to the list of addresses and AddressList is a linked list of addresses. A broadcast capability to all nodes on a given LAN or to all nodes on the network is provided by supplying a "broadcast address" in the first form of the SEND command shown above. The ability to scan the incoming datagram queue for messages with a specific combination of transmission parameters (e.g., priority, time-tag, etc.) is provided by the SCAN command which returns the message count (number of messages) with this combination of transmission parameters.

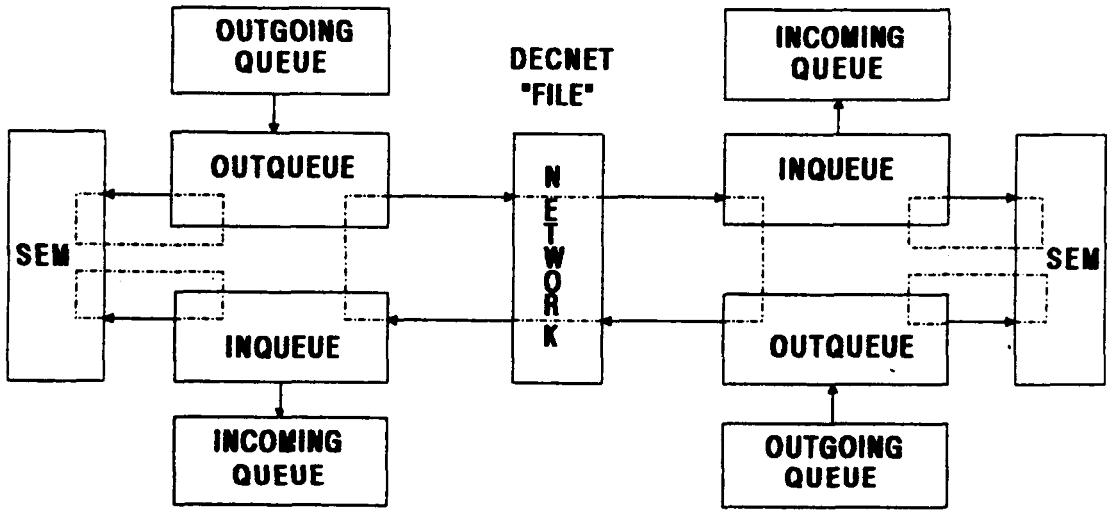


Figure 4

A message can be physically retrieved from the incoming queue using the RETRIEVE command. The RETRIEVE utility provides selective retrieval for a speci fic combination of transmission parameters or it can be used without parameters to retrieve the highest priority message in the incoming queue.

Virtual Circuit Service Simulation

When the information exchange between subsystems of the DMS must simulate a continuous dialogue over a physical circuit or when near real time transmissions are necessary it is usually more efficient to establish a "virtual circuit" between the subsystems. Some of the advantages of a DMS virtual circuit are:

(1) the routing information (source and destination address) does not have to be provided to the NOS for every transmission,
(2) network bandwidth is reserved for the dialogue, insuring a certain maximum transmission delay and
(3) messages are always received and handled in the same sequence they are transmitted.

The simulation of the virtual circuit capability on the TAVERNS Emulator is very similar to that of the datagram service. The main difference is that dedicated incoming and outgoing queues are established for each circuit in both the subsystem requesting the connection and the subsystem being connected. The other main difference is that virtual circuit traffic is "multiplexed" over the DECNET connection to simulate the reservation of bandwidth but the ability to assure a maximum transmission delay of the order of magnitude anticipated for near real time communications on the station is not possible using DECNET when other users are on the system.

To establish a connection (virtual circuit) an Ada subprogram or task calls the CONNECT procedure and to deallocate a circuit (and its associated queues) the DISCONNECT procedure is used. Once a connection is established the connected Ada subprograms can transmit and receive using the XMIT and RECV commands whose formats are:

XMIT(MESSAGE,CIRCUIT); and RECV(MESSAGE,CIRCUIT);

where MESSAGE is the Ada object to be transmitted and CIRCUIT is the circuit number assigned to the virtual circuit by the NOS at the time of connection. The Ada procedures that provide the virtual circuit capability are contained in the generic package VIRTUAL (this does not include the connection service). An Ada subprogram using the virtual circuit capability must instantiate a version of this package for each different Ada object that will be transmitted.

Data Acquisition and Distribution Services Simulation

The DMS Data Acquisition and Distribution (DAD) Service is layered over the datagram and virtual circuit services provided by the DMS. Which of these underlying services is used depends upon whether the usage of the DAD service is periodic or not (as will be explained below). As previously mentioned, one of the key features of the DMS Services is the ability to request readings of sets of measurements. The simulation of this service in the TAVERNS Emulator is based on the following (simplified) Ada measurement object defined by:

```
type MEASUREMENT(REP: RepType; d1, d2, d3: positive) is
  record
    NAME: string(1 . . 15);
    case REP is
      when FLOATPNT     => FVAL : float;
      when FIXEDPNT     => IVAL : integer;
      when TEXT         => TXT  : string(1 . . d1);
      when FLOATARRAY1 => FVAL1: FLTARRAY1(1 . . d1);
      when FLOATARRAY2 => FVAL2: FLTARRAY2(1 . . d1, 1 . . d2);
        :
        :

    end case;
  end record;
```

where the discriminant REP is a variable of the enumeration type RepType that includes an entry for each data structure that the Data Acquisition and Distribution (DAD) Services supports and where d1, d2 and d3 are parameters that indicate the size of arrays, strings, etc. to be associated with a measurement as its "value".

Measurements are known globally by their NAME which is recorded in the NAME field of the MEASUREMENT variant record object. Each measurement is owned by some subsystem and this ownership is known to the DMS Services. The value of a measurement is stored in the variant part of the MEASUREMENT record object and can be of any type for which a corresponding entry in RepType exists. The enumeration list for RepType shown above is only representative as many of the types are yet to be determined.

The package DATAREQUEST contains the procedures for preparing a request message for transmission. They are REQUEST, MAKESET, GETSET and READNEXT. The MAKESET procedure associates a name with a list of measurements. The format of the MAKESET command is then defined by

MAKESET(SetName,MeasurementList)

where SetName is the name to be assigned to the measurement set and MeasurementList is a list of measurement names. The REQUEST command requests the reading of a set of measurements. The format of the REQUEST command is:

REQUEST(SetName) or REQUEST(SetName,PERIOD)

The second form of this command utilizes the PERIOD parameter which is a request for a periodic reading of the measurement set every PERIOD seconds where PERIOD is a non negative floating point number. The following steps outline the requesting procedure.

(1) call the MAKESET procedure to create the named measurement set,
(2) call the REQUEST procedure to request the reading of the set,
(3) call the GETSET procedure to obtain the set name of the next measurement set that has been successfully processed as a result of a previous REQUEST and
(4) use the STATUS parameter of the GETSET command to determine if the reading of a set has been completed.

The format of the GETSET command is:

GETSET(SetName,STATUS,TIME)

where SetName is the name of a previously requested measurement set whose processing has been completed at the time returned in the TIME parameter. Once a set name is obtained from the GETSET utility, the measurements in the set can be read using the READNEXT procedure whose command format is:

READNEXT(SetName,MEAS)

where MEAS is of type MEASUREMENT. To use the measurement's name as an operand is straightforward since its type is known to always be a 15 character string. To perform an operation on a measurement's value, however, involves examination of the discriminant of MEAS since MEAS is a variant record. The STOP command is used to stop an active periodic REQUEST. The format of this command is:

STOP(SetName).

The DATAREPLY package provides the necessary procedures to be used in responding to a REQUEST. These utilities are: REPLY, SETNAME, NEXTNAME, and WRITENEXT. They are similar in nature to the utilities provided in the DATAREQUEST package.

N89-16367

A STUDY OF THE USE OF ABSTRACT TYPES FOR THE REPRESENTATION OF ENGINEERING UNITS IN INTEGRATION AND TEST APPLICATIONS

Charles S. Johnson

ABSTRACT

Physical quantities using various units of measurement can be well represented in Ada by the use of abstract types. Computation involving these quantities (electric potential, mass, volume) can also automatically invoke the computation and checking of some of the implicitly associable attributes of measurements. Quantities can be held internally in SI units, transparently to the user, with automatic conversion. Through dimensional analysis, the type of the derived quantity resulting from a computation is known, thereby allowing dynamic checks of the equations used. Through error analysis, the precision with which a quantity is measured can be correctly propagated into the result of a computation involving that quantity. The output of both measured and computed quantities can automatically be rounded to the correct significance, and labeled with the correct units.

The impact of the possible implementation of these techniques in integration and test applications is discussed. The overhead of computing and transporting measurement attributes is weighed against the advantages gained by their use. The construction of a run-time interpreter using physical quantities in equations can be aided by the dynamic equation checks provided by dimensional analysis. The overhead of responding to measured and computed system variables in real-time systems can be decreased in the case where only the significant changes in data values are responded to. The effects of higher levels of abstraction on the generation and maintenance of software used in integration and test applications are also discussed.

INTRODUCTION

Data abstraction should, in the near future, become the most important tool used in the Ada development of replacements to current systems functioning in the area of Integration and Test (I & T). This importance stems from the urgent need to maintain Test Procedure/ Test System Independence. This independence promotes both the reusability of Test Procedures and the possibility of modifying physical device information in the Test System, at run-time, without affecting procedures using logical access methods. This is necessary to decrease turn-around time due to modifications of the Test System/ Test Article hardware configurations.

BRIEF BACKGROUND

Kennedy Space Center/ Engineering Development/ Digital Electronics Engineering Division is in the process of prototyping distributed systems supporting I & T applications, particularly the Space Station Operations Language (SSOL) System, which is the I & T subset of the User Interface Language (UIL) for the Space Station. The discussions in this paper were developed from the results of systems designed and developed in Ada to demonstrate the feasibility of supporting the abstract data types used in I & T, specifically, engineering units. The Ada environment used was that of VAX Ada under VAX/VMS.

SYSTEM CONCEPT

There is a direct correlation between the effectiveness of computer systems and the fidelity with which objects in those systems simulate the behavior of the external phenomena that they are intended to represent. [1] The definition of objects is then akin to a simulation effort: complete with objectives outlining progress towards simulation goals, and constraints which limit the scope of the effort.

The goal of object definition for measurements and quantities used in Integration and Test applications is to create objects representing the physical quantities that are measured, tracking a magnitude for the quantity, and the type of quantity. The quantities (VOLTS, METERS/SECOND, PSI) should interact with other quantities in the same way that real physical phenomena do:

$$V = IR$$
$$PV = nRT$$

In other words, arithmetical operations should convert the quantities correctly into new quantities. Also it would be useful if the creation, input, and output of those quantities could be performed using any unit or scale of measure (length in METERS or MICRONS or CUBITS). It would be nice, as well, to know the precision with which a measurement was made, so that it can be determined if it represents a significant change from the last measurement. That precision, or measurement error, should propagate correctly during computation as well.

The objectives which mark progress towards these goals can be established. The quantities and units should be easy to define and use. The quantities should convert correctly upon input in different units. The quantities should convert correctly upon computation, and if the resultant quantity is of the incorrect type, an exception should be created, because the equation is incorrect (or the result type is wrong). Precision should be computed correctly for the different arithmetic operations. Finally, if the wrong units are selected for input or output, an exception should be generated.

The constraints which confine the scope of the effort can be defined. It is important that the support of the features of the system should not incur excessive system processing or storage size overhead, because too much time and space costs money (space less than time these days). The resulting packages should not be too complex, relying instead on algorithms and structures that are just complex enough to create a useful result. Lastly, the development should be constrained against passing the point of diminishing returns. If a feature is difficult to implement and yields little in tangible results, it should be forgone.

PHYSICAL QUANTITIES

The tracking and converting of types of quantities is simply and efficiently done in computer programs by dimensional analysis. [2] This involves some fairly simple physics, for example, the average acceleration of a body can be computed by the equation:

$$\text{Average Acceleration } (m/s^2) = \frac{\text{Velocity Change } (m/s)}{\text{Time of Change } (s)}$$

which uses the units m = meters and s = seconds. The average force applied to the same body can be computed by:

$$\text{Average Force } (N) = \text{Mass of Body } (Kg) * \text{Average Acceleration } (m/s^2)$$

which uses the units N = newtons and Kg = kilograms as well as meters and seconds. What can be seen from combining the units of these equations is the following units equivalency:

$$\text{newtons } (N) \cong \frac{\text{kilograms } (Kg) * \text{meters } (m)}{\text{seconds squared } (s^2)}$$

The units like kilograms, meters and seconds are called base units, and the units like newtons are called derived units. These are all SI units, standardized by the ISO Resolution R1000 in 1969, and documented in the Le Systeme Internationale de'Unites (BIPM), but conversions exist for all other forms of units as well. If a matrix of derived units versus their base units is made, the dimensionality of derived units in their base units can be shown (Table F.4.4-1). The newtons unit shows a one in the kilograms column, a one in the meters column and a -2 in the seconds column, because seconds squared is reciprocal.

This dimensional analysis can be done for most units in any of the systems (English, CGS, etc.). Some units, however, are truly dimensionless. An example is the decibel and the Richter scale units, which are logarithms of ratios of units which cancel out. Some units just do not fit into dimensional analysis. AC circuit impedance equations do not cancel nicely, for instance, and AC units would probably have to be defined as dimensionless for those equations to correctly cancel. This simple dimensional analysis, as a whole, probably deals badly with sinusoidal phenomena.

TABLE F.4.4-1: PARTIAL TABLE OF UNIT DIMENSIONALITY

For each unit, the dimensionality is given versus each base unit from which it is derived (meter, kilogram, second, ampere, kelvin, candela & mol), along with the scale and offset required (1.0, 0.0 for SI derived units).

	Base Units							SI Conversion
Derived Units	m	Kg	s	A	K	cd	mol	Scale, Offset
newton:N	1	1	-2					1.0, 0.0
hertz:Hz			-1					1.0, 0.0
joule:J	2	1	-2					1.0, 0.0
watt:W	2	1	-3					1.0, 0.0
volt:V	2	1	-3	-1				1.0, 0.0
lumen:lm						1		1.0, 0.0
henry:H	2	1	-2	-2				1.0, 0.0
molarity:M	-3						1	1000.0, 0.0
astron. unit:AU	1							1.496E+11, 0.0
footpound:ft-lb	2	1	-2					1.356, 0.0
knot:kt	1		-1					0.5144, 0.0
slug		1						14.5939, 0.0
fahrenheit:^{o}F					1			0.5556, 255.37

The advantages of this method of tracking dimensions are mostly in verification of physical equations used in I & T applications. Even very complex equations involving many factors can be analyzed. During addition and subtraction operations, the two input quantities and one output quantity must be identical dimensionally. During multiplication the dimensions are added, and in division they are subtracted. If the result type doesn't match the computed dimensionality, it is an error. Dimensions can be stored as integers of range -20..20, and the overhead involved in integer arithmetic and compares is probably little.

The disadvantages are that it doesn't deal well with AC quantities and the like, which would require a complicated and unwieldy solution, yielding few tangible returns. Also, there are several correct dimensional solutions, any of which can be misapplied to a problem, with no detectable dimensional error (series/ parallel DC circuit equations).

MEASURED PHYSICAL QUANTITIES

The measurement of physical quantities always incurs a measurement error which can be assigned to the measured quantity at it's source, as it enters the system. This precision is key to any analysis of the significance of the measured quantity. If two sequential measurements of the same phenomena are obtained, and their difference is less than the precision by which they are measured, then there is no significant difference, and the measurement is considered the same, experimentally. The scope of use for measurement precision would then be anywhere, in the system, after the conversion from raw data (counts) into engineering units. It should be noted here, that the measurement precision analysis discussed is different from the significant change analysis used in the front end processing of raw counts in the Launch Processing System (LPS), which is a digital process for raw data concentration to remove line jitter.

Error propagates (increases) as measurement values are combined by physical equations to yield resultant quantities. If the precision is an available attribute of the measured quantity, then the precision of all computed quantities can likewise be computed and carried along with the measurement. The computation of propagated error from the mathematical operations applied to quantities is shown in Table F.4.4-2. The relative-type error, on the right, looks like it will produce many occasions of division by zero, and is therefore not useful. The absolute-type of error, in the center, looks to produce a divide by zero only when the operation is a divide by zero (in error), and seems optimal.

For quantities introduced into the system without a measurement, such as constants, the precision input would be derived from the number of significant digits (+/- one half of the last digit).

TABLE F.4.4-2: FORMULAE FOR LIMITING ERROR

For the following mathematical functions: f(x, y); given that x and y are the exact values, a and b are their measured approximations, and the deltas for a and b are their limiting errors.

Type Of Function	Bounds For The Absolute Error	Bounds For The Relative Error
$x+y$	$\triangle a+\triangle b$	$(\triangle a+\triangle b)/\lvert a+b\rvert$
$x-y$	$\triangle a+\triangle b$	$(\triangle a+\triangle b)/\lvert a-b\rvert$
$x*y$	$\triangle a\lvert b\rvert + \triangle b\lvert a\rvert$	$\triangle a/\lvert a\rvert + \triangle b/\lvert b\rvert$
x/y	$(\triangle a\lvert b\rvert + \triangle b\lvert a\rvert)/b^2$	$\triangle a/\lvert a\rvert + \triangle b/\lvert b\rvert$
x^n	$\triangle a\lvert n a^{(n-1)}\rvert$	$\lvert n\rvert(\triangle a/\lvert a\rvert)$

DERIVED TYPES SOLUTION

The simple object-oriented approach to measured quantities would be to consider units to be classes of measurements, and to make them derived types of a base record type which would have the components mentioned: measured value and precision, and dimensional values. Then the combinations of these types would be performed by defining, for example, a multiply function that takes inputs of AMPS and OHMS and makes VOLTS. To define the legal combinations for just a few types would be laborious, there are just too many relationships. The simple approach is too complex.

DISCRIMINANT TYPE SOLUTION

A solution for the representation of physical quantities using discriminant records is pointed to in Hilfinger [3]. It is not written exactly in Ada, though, for he presents a case for possible changes in the language. The record discriminants are the dimension values and the units scale factor, which would then prevent assignment of dissimilar units of the same dimensionality. For example, quantities in meters could not be assigned to quantities in feet, although the dimensionality is the same. Assignment of dissimilar constrained records is then accomplished by the overloading of the assignment operator ":=", with a function that re-scales the internal value to the new scale factor, and creates the correct and matching constraint values.

In current Ada, however, only discrete discriminants are legal, which disallows units scale factor as a discriminant (because it is a real type), and the ":=" operator cannot under any circumstances be overloaded. So it doesn't work in standard Ada.

An attempt can be made to standardize that approach, but there are some problems without the fixes to Ada. If the scale value were kept as a record component, instead of a discriminant, it will be modified upon assignment (not a constraint anymore). This negates the ability to keep scale in the quantity, and the quantity scaled as feet, instead of meters.

If the scale for an engineering unit alone is kept, then offset units, such as degrees fahrenheit (not aligned with absolute 0 ^{O}K) cannot be used.

There is also unnecessary run-time overhead to re-scaling every time a computation is made, and possible rounding error in the scale, which may drift. The rounding error in the scale is probably the reason why Ada doesn't allow it or any other real type to be used as a record discriminant.

CLOSELY-COUPLED DISCRIMINANT TYPES SOLUTION

A further redefinition of an object can be accomplished with differentiation [1], when the object definition has become too amorphous to simulate the target phenomena. Differentiation could be considered a fine structure definition technique for systems, whereas Object-Oriented Design or Functional Decomposition are gross structure definition techniques.

A separation of the object definition for physical quantities is made, into two closely-coupled objects, QUANT and

UNIT. QUANT is the measured quantity, and UNIT is one possible engineering unit for a quantity. They have attributes in common, the dimensional values. They also have unshared attributes.

UNIT is a private dimensionally-constrained discriminant record which contains the scale and offset for the engineering unit it represents, and can have for components other engineering unit attributes, such as the text label for output functions, or an input prompt text.

QUANT is a private dimensionally-constrained discriminant record which contains the measured value stored in SI units (no re-scaling), and can have for components other measurement attributes, such as measurement precision or identification of source device or process.

Arithmetic interactions of real types with type QUANT should be similar to those between scalars and vectors, only multiplication and division being allowed for scaling the QUANT values. Arithmetic interactions of real types with type UNIT should be similar to those between scalars and unit vectors, and therefore a QUANT is the outcome. Any arithmetic interaction of a UNIT with a UNIT or a QUANT should produce a QUANT, converting the pure to the impure, so to speak. QUANT objects should arithmetically combine to produce QUANT objects, of course.

With these definitions for the private types and arithmetic functions, it is simple to define several QUANT subtypes for the physical quantities (LENGTH, MASS, VELOCITY, POTENTIAL, WORK, INDUCTANCE, etc.) and to define several UNIT constants (deferred constants in the package) for the engineering units (FT, KG, KPH, VOLTS, FT_LB, HENRYS, etc.). It should be simple to create values for QUANT on the fly:

```
PIPE_LENGTH : LENGTH := 5 * FT;
GAS_CONSTANT : CONSTANT QUANT :=
                       8.31434 * JOULES / ( DEG_K * MOLS );
```

Functions for creating new UNIT constants on the fly will be necessary, since they cannot be produced arithmetically or defined externally to the package. I/O functions for QUANT values will also require a UNIT constant as a parameter, for scaling to/from SI units. A function for extracting the value of a QUANT object as a real variable, will also require a UNIT parameter and a conversion.

It would be possible, with a private dimensionally-constrained discriminant record variant, to create one type by lumping both QUANT and UNIT attributes together (one discriminant chooses which). This, however, is an unsatisfactory technique. With variant objects, the programmer always has to check what he has, before he can use it. The overhead of such checking is little, but the complication is now pushed into the application, instead of being in the package. This would seem to be a reversal of the purpose of abstraction.

Measurement precision could be included as a component in the QUANT definition by the use of the absolute precision computations listed in Table F.4.4-2. The absolute quantity precision would then be computed into any result, like the dimensionality and the measured value itself.

The offset component of the UNIT type, could be used for more than just offset temperature scales (celsius, fahrenheit). Any differential scale could be represented by an engineering unit. In an example, cargo positional coordinates could be internally held in a centralized coordinate scheme. Differential voltages or pressure readings from sensors could also be related to some reference point.

If the pre-defined UNIT constants were ordered into a table, the I/O functions could, given a quantity of unknown type, select an output label and scaling, or an input prompt. This might be particularly useful in the generation of reports or ad hoc queries, which would use computation involving quantities and creating new quantities on the fly.

USE OF DISCRIMINANT TYPES IN GENERICS

Along with the useful constraining features of discriminant records, comes the difficulty of matching them with generic formal parameters. To instantiate a generic software component with a formal parameter matching a discriminant private type, the type must be constrained (no unconstrained types in generics), and the type of constraint passed as a generic formal parameter first. Then the discriminant type is passed, as a discriminant generic formal parameter. It is fairly obvious that most generic software will be produced, not of this type, but using the type private, with accompanying functions of that type (as in the generic sort function in most textbooks).

This problem can be handled, for any unconstrained discriminant type (QUANT, UNIT) with constrained subtypes (VOLTAGE, POWER), by declaring a non-discriminant record type which contains the unconstrained discriminant type. To instantiate a generic sort function, the enclosing record type would have to be passed to the generic, for the creation of an array type (for sorting), and an ordering function ">" for the enclosing type would also have to be defined and passed.

This is somewhat of a kludge, in that the constraints do not apply within the scope of the generic component. Simply put, the discriminant types feature of Ada somewhat precludes the use of generic software in a straightforward manner.

ANALYSIS OF OVERHEAD FOR USAGE OF OBJECT DEFINITION

The storage overhead can be estimated on the assumption of bytes for dimensional integers and 4-byte floating point representation for the measurement itself and for it's precision. This gives a 2X storage increase for carrying the dimensions and another 1X for the precision, up to 4X for everything. In communications, with all of the rest of the overhead involved in sending a measurement in a packet, this probably is not significant (other measurement information, status, device status, send/receive addresses, transaction ID, packet ID, etc.).

The computational overhead for the dimensional analysis feature, which uses integers, is thought to be small compared to the floating point math involved in each multiply and divide for the measurement itself. This is thought even though there are seven dimensional integers being added for every measurement being multiplied (inversely for divides). Measurement adds and subtracts simply involve comparing for the dimensions (can't add VOLTS to WATTS).

The computational overhead for the precision feature, if absolute error is propagated by a floating point representation, is about 1X for adds, subtracts and the power function, 2X for multiplies, and 4X for divides. This can be seen in the central column of Table F.4.4-2.

ADVANTAGES OF DIMENSIONAL ANALYSIS FEATURE

The low computational overhead incurred by this feature is more than compensated by the advantages it carries. These are in the area of verification, validation, and run-time interpretation support.

During the development of I & T software, the use of constrained types to represent quantities should make possible the verification by dynamic analysis of that software. Even the most complex equations using dimensional variables, can be checked for the correct and allowable combination of subtypes, and for the return of the correct types of physical quantities. However, this will not catch those mistaken computations which return the correct quantity type, incorrectly computed.

The dimensional analysis method, since it is a dynamic feature of programs using it, and not a static feature, will lend itself well to validation of programs as well. In large I & T applications (for example LPS), the binding of logical measurement designation to physical device parameters is delayed for as long as it is possible. This allows the modification of hardware parameters with the minimum impact on the software system. The optimum circumstance would involve run-time binding of the logical level to the physical, so that the hardware configuration could be changed at test-time without having to patch the system, as is done now.

In that desired situation, there will be a large separation between the analysis of the logical nature of a program (equations), which would occur during development and verification, and the physical validation of the program against the model, or components of the Test System. As the distance between the verification of the logical and the validation of the logical-to-physical widens, the potential for dynamic problems to escape unnoticed should increase. If methods for logical verification of programs at run-time are used, such as dimensional analysis of equations by the method proposed, the possibility of catching these dynamic problems increases.

This problem of run-time dynamic analysis is exacerbated in the use of I & T command languages such as the User Interface

Language (UIL) and it's subset, the Space Station Operations Language (SSOL). The commands in these languages are interpreted at run-time, and are formulated by the user at the terminal, on-orbit. The use of complex equations in these languages to perform control functions is proposed. Syntax checking can be performed easily by the User Interface, but checks for correctness of the physical equations used will require some facility, such as dimensional analysis. If dimensional analysis were used, the internal checks in the interpreter program would be automatic against every statement using physical quantities, and exceptions would be generated on a statement by statement basis.

ADVANTAGES OF COMPUTATION OF PRECISION FEATURE

As the complexity of systems increases, leading up to the Space Station era, so does the number of levels of integration to be passed through by components before their operational use. In some shuttle payloads from ESA, there are already 7 levels of integration. If Ada is to become widespread in it's use as the I & T language supporting these levels of integration, then the Ada software products must be promotable between levels of integration. This does not imply a need to run the same procedure at a higher level, although that may be a requirement. What it does require is that lower level software components be incorporated by some method of abstraction into higher level components, up through launch operations and on-orbit operations.

At each level of abstraction, component level state, control and measurement variables are presented as parameters to higher-level integration software, simplifying interfaces at the subsystem, and then the system level. This continues until at the on-orbit user interface level, simple designators for systems are connected to a large tree of state, control and measurement variables extending all the way back down the integration chain.

In systems using abstracted measurement variables, knowing the significance of measurements at all levels is an important issue. Control logic algorithms which attempt to establish set points to an insignificant range are erroneous. Commands which effect an insignificant change in an effector are meaningless, and consume system resources in their performance. Measurements which involve insignificant changes in levels should not be communicated.

Communication overhead becomes more of an issue, as we progress from tightly-coupled shared-memory systems (like LPS), to loosely-coupled distributed systems (like GDMS prototypes). Distributed systems have failure modes related to communication loading (traffic jams), which can be abated somewhat by data concentration.

Concentration of data at the very lowest level of measurement has, and will probably continue to be performed on the raw data by bit-oriented algorithms. After the basic measurements have been converted to engineering units, however, there are data concentration possibilities, based on significance and propagated significance, that affect communications, and the stimulus and response of the system.

GENERAL ADVANTAGES OF DATA ABSTRACTION APPLY

The systems supporting the future I & T applications described in the last section, will be highly distributed. They will contain components from several levels of integration and will also need to be programmed at the highest level possible. The programs which drive the higher-level system functioning should not be bogged down with detailed data analysis. Facilities supporting the propagation of information concerning the validity of measurements and the validity of algorithms concerning those measurements should be basic to the system. Complex programs integrating the functioning of a distributed I & T system will be inherently more maintainable and reusable if kept at highest possible level of data, system and resource abstraction. Greater readability and verifiability of software components, and greater reliability and ease of validation of the system code is then possible. Finally, the design and development of the user interface level applications becomes easier, the higher the level of abstraction that is achieved for the system components and measurements.

ACKNOWLEDGEMENT

I gratefully acknowledge the support given by the Kennedy Space Center/ Engineering Development/ Systems Integration Branch in supplying the computer facilities for the feasibility studies that provided the basis of this work. I also thank my wife, Bronwen Chandler, for her support.

REFERENCES

1. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station.

2. Karr, M., and Loveman, D. B. III. May 1978. "Incorporation of Units Into Programming Languages", Communications of the ACM, Vol. 21, No. 5.

3. Hilfinger, P. N. 1983. Abstraction Mechanisms and Language Design. Cambridge, Massachusetts: The MIT Press.

N89-16368

Rdesign: A Data Dictionary with Relational Database Design Capabilities in Ada[*]

Anthony A. Lekkos[1]
Teresa Ting-Yin Kwok

University of Houston, Clear Lake

1. Introduction

Data Dictionary is defined to be the set of all data attributes, which describe data objects in terms of their intrinsic attributes, such as name, type, size, format and definition. It is recognized as the database for the Information Resource Management -- to facilitate understanding and communication about the relationship between system applications and system data usage and to assist in achieving data independence by permitting system applications to access data without knowledge of the location or storage characteristics of the data in the system [Allen82].

The following are considered to be its primary objectives:-

1. To achieve control of the data resource, by providing an inventory of that resource. To enforce standards and validation.

2. To control the costs of developing and maintaining applications.

3. To provide for independence of metadata across computing environments, improving resiliency to the effects of hardware and software changes [Allen82].

Much of the importance of a data dictionary has been recognized, yet, little of it has been utilized to support an automated database design.

[*] Ada is a registered trademark of the U.S Government- Ada Joint Program Office.
[1] Supported by NASA/JSC-UHCL Ada-Beta site Contract.

A research and development effort to use ADA at UHCL has produced a data dictionary with database design capabilities. This project supports data specification and analysis and offers a choice of the relational, network, and hierarchical model for logical database design. It provides a highly-integrated set of analysis and design transformation tools which range from templates for data element definition or modification, spreadsheet for defining functional dependencies, normalization, to logical design generator.

2. The Data Dictionary with Database Design Capabilities

2.1 The Data Dictionary

The structure for the data dictionary is essentially relational in nature with the data element definition normalized to third normal form, while the related projects are kept in another relation. Further, the dictionary is furnished with the following facilities:-

Define --
: creates a new data element entry in the data dictionary. A template is used to enter the data element name, type, size, range, description, validation rules, picture, intensity, display attribute, should the element type be enumeration, the enumeration list could also be entered.

Modify --
: changes any data element specification created with "Define". If the data element name is changed, it creates a new data element under this new name -- essentially, it performs a "Copy" function under this circumstance. Again, a template is used for all field entries.

Search --
: retrieves data element specifications from the data dictionary and displays them on screen. This differs from "Report" in that only data element names are displayed. A global search could be done by entering an "*". A list of names will be displayed on screen one page at a time.

Purge --
removes a data element from the data dictionary if the data element is used only in the current project. The element is not purged if it is used by other projects.

Transfer --
imports data element definitions from an external text file or exports data element definitions to an external text file. All of the elements could be imported or exported all at once, or they could be imported or exported individually, or they could be imported or exported according to projects.

Report --
lists in detail the data element definition for a single data element or a series of data elements. The listing could go to the terminal or to the system printer. If terminal is chosen as the output device, the data element definition will be displayed on screen one page at a time.

2.2 Functional Dependencies

Given a project with its own set of data elements, one can proceed to define functional dependencies amongst data elements. The following facilities are provided :-

Clear --
which clears out all previously defined functiona dependencies.

Spreadsheet --
data element names are displayed in rows and columns in a spreadsheet. Entering appropriate symbols in corresponding positions or "cells" will define functional dependencies amongst elements. Using the tab key or arrow keys, one can move around the cells. Should the arrow go beyond bounds the spreadsheet will move one column left/right or one row up/down dependent on the arrow key hit and its position. One can also move the spreadsheet one page at a time by pressing Function key 1, 2, 3 or 4 to go up, down, left or right.

The following symbols are used to define functional dependencies :-

==> means row element determines column element

<== means column element determines row element

KEY means element is a key, row element name should be the same name as column element

N/A means not applicable, row element name should be the same as column element name where the element is not a key

+=> means concatenation of row elements to identify column element

<=+ means concatenation of column elements to identify row element

To make the spreadsheet even more convenient to use, there are a few hidden keys :-

R = Refresh --
the screen is refreshed, in addition todisplaying symbols used to define elements functional dependencies, the complementary symbols are also displayed.

H = Help --
help can be invoked.

B = Beginning --
spreadsheet moves to the beginning of the list of data elements row-wise or column-wise.

E = End --
spreadsheet moves to the end of data element list row-wise or column-wise.

F = Find --
gets a particular data element which will be displayed in the middle of the list row-wise or column-wise.

T = Toggle --

Toggles the symbol, for example, pressing "T" at a place where it displays "N/A" will change the symbol to "KEY".

To update the functional dependencies, one only needs to blank out the entry, enter appropriate symbols or just toggle the symbols.

P = Print --

will print out the functional dependencies to the screen or to a file.

2.3 Database Design Generator

After the functional dependencies are defined, the normalization tool can be utilized to automatically normalize the relations in third normal form. Each table structure is displayed and a name should be given.

At this point, all the tables so created are in third normal form.

For application or implementation reasons, one may have to violate the rules for normalization or to keep certain relations not in third normal form. A "maintain-table" facility is provided so that a database designer can define his own table with its own set of keys and attributes. Moreover, he can rename a table, delete a table, delete certain keys or attributes in a table or add certain keys or attributes in a table. The system will not re-normalize these tables. There is one constraint, however, the keys and/or attributes had to be defined in data dictionary.

Again a spreadsheet is employed to define the relationships amongst relations, be it one-to-one, one-to-many, many-to-many or no-relations.

A refresh function will not be applicable in this case as the relationship between the row relation and the column relation may not be reciprocal.

A parent-child graph could be generated after the relationships are defined. The graph could be printed out to the system printer or to the screen.

The conceptual schema is then generated and output goes to the screen and a text file so that the designer can view it and make modifications if necessary. Since the data elements used are governed by the data dictionary, consistency, integrity and validity can be achieved easily.

Following is a SQL-type logical design interface so generated :-

```
Rem
Rem   SQL/DS Database Design Identification Section
Rem
Rem   Application : DIS
Rem
Rem   Date created: 6/6/86
Rem
Rem   ----------------------------------------------------
Rem   SQL/DS Database Tables Create Commands Section
Rem   ----------------------------------------------------

Create Table DEPT
     (DEPT_NAME            Char      (32) not null,
      BUDGET               Number     (7),
      DEPT_MGR             Char      (32),
      LOCATION             Char      (32));

Create Unique Index DEPT_INDEX on DEPT (DEPT_NAME);

Create Table EMPLOYEE
     (EMP_NAME             Char      (32) not null,
      DEPT_NAME            Char      (32),
      EMPLMNT_DATE         Char       (8),
      POSITION             Char      (32),
      SALARY               Number     (7));

Create Unique Index EMPL_INDEX on EMPLOYEE (EMP_NAME);

Create Table PROJECT
     (PROJ_NAME            Char      (32) not null,
      DEPT_NAME            Char      (32),
      CHARGE_NO            Number     (4),
      COMPL_DATE           Char       (8),
      PROJ_LEADER          Char      (32));

Create Unique Index PROJ_INDEX on PROJECT (PROJ_NAME);

Create Table EMP_PROJ
     (EMP_NAME             Char      (32) not null,
      PROJ_NAME            Char      (32) not null,
      CHARGED_HRS          Number     (4),
      EMPPROJ_DATE         Char       (8));

Create Unique Index EMPPROJ_INDEX on EMP_PROJ
     (EMP_NAME,PROJ_NAME);
```

3. User Interface

Much of the work for maintaining the data dictionary is done through a template, e.g., define and modify, user only needs to fill in the blanks, other work like defining functional dependencies and relationships amongst tables is done through a spreadsheet. The whole system is menu driven with the first two rows of the screen dedicated to commands. To go up to the command line, one only needs to press <F2>. One can then use Tab, or arrows to move across the command line. A <return> selects th command. The 24th row on the screen is dedicated to function key explanation while the 23rd row is used for message line and the 22nd row is used for prompt line. The rest of the screen will be used for display or for the template. Help could be invoked throughout the screens.

References

[Allen82] Frank W. Allen, Mary E. S. Loomis, Michael V. Mannino "The Integrated Dictionary/Directory System" , ACM Comp. Survey,14:2, 1982.

[Goldstein85] Robert C. Goldstein "Database :Technology and Management John Wiley and Sons, Inc., 1985.

[Curtice84] R.M. Curtice "IRMA: An Automated Logical Data Base Design and Structured Analysis Tool", IEEE Database Eng. 7:4, December 1984.

[Reiner86] David Reiner, Gretchen Brown, Mark Friedell, et al, "A Database Designer's Workbench" submitted to Dijon ER Conference, 1986.

[Bjornerstedt84] A.Bjornerstedt and C. Hulten "RED1: A Database Design Tool for the Relational Model of Data", IEEE Database Eng. 7:4, December 1984.

N89-16369

Ah!Help: A Generalized On-line Help Facility

Wong Nai Yu*
Charmiane Mantooth*
Alex Soulahakil*

1. Introduction

In modern-day programming it is not sufficient that a large, commercially available, software package simply work. It must have certain characteristics that make it marketable. Among these, user friendliness and portability are of major importance.

In the user friendliness category, a great forward step was taken with the introduction of on-line help facilities, relieving the user from continually turning to cumbersome manuals for assistance. The on-line help facility we have designed is neither unique nor revolutionary. It is a simple program which was originally designed to work in conjunction with a screen generation package. It is, however, independent of it and is, therefore, portable. The only new aspect introduced by this package is that it is written in Ada and utilizes certain Ada facilities, such as binary files and Direct_Io, which make implementation neater, simpler and more straight-forward than languages have heretofore allowed. In addition, the program uses only standard Ada generics, thus adding to its portability.

The concept behind this package is to allow the building of help files in textual format. The program then builds a binary file, creating and storing an index for the named file. This index, along with secondary indices created for further help on specific choices made available to the user, is later used to access the help file associated with the program currently being used by the user. Upon exiting the help mode, the user is returned to the point from where the on-line help was requested.

2. Design Considerations

The basic and overall purpose of this program was to produce a convenient, easy to use, general purpose on-line help facility. Convenience and ease of use from the end-user's point of view, however, usually means time consuming, difficult programming for the low level designer. It also generally means that there is a loss of generality and/or portability.

In order to combine these seemingly incongruent features we decided that an implementor/designer should be able to provide an on-line help menu and associated verbal descriptions in textual format, via a text file. The program should take over from there and create the actual manual which the user sees. The text file method does, however, present certain restrictions. It is highly formatted, meaning the designer of an on-line help manual must be familiar with the rules imposed by the program. We felt this to be only a minor inconvenience in comparison to the advantages to be gained by the program.

The main design considerations were that the user should, by use of a function key, be able to call an on-line help facility pertinent to whatever mode/program he or she happened to be in.

```
 _________________________________________________
|        PAYROLL / PERSONELL SYSTEM               |
|                                                 |
|        SOCIAL SECURITY NUMBER:                  |
|        LAST NAME:                               |
|        FIRST NAME:                              |
|        MIDDLE INITIAL:                          |
|        STREET ADDRESS:                          |
|        CITY AND STATE:                          |
|        ZIP CODE:                                |
|                                                 |
|_________________________________________________|
```

USER PRESSES F1 FOR HELP

```
 _________________________________________________
|                                                 |
|    _______________________________________      |
|    | GENERAL COMMANDS => FUNCTION KEY CALLS |   |
|    | GENERAL PANEL DESCRIPTION              |   |
|    | PANEL ATTRIBUTE DESCRIPTIONS           |   |
|    ------------------------------------------   |
|                                                 |
|    -------ACTIVE COMMAND MENU---------------    |
|    |  USE RETURN TO RETURN TO PANEL        |     |
|    ------------------------------------------   |
|                                                 |
|    ------------------------------------------   |
|    | USE ^U TO MOVE UP, ^D TO MOVE DOWN,    |   |
|    |     RETURN TO SELECT OPTION            |   |
|    ------------------------------------------   |
---------------------------------------------------
```

USER USES ^D TWICE AND SELECTS PANEL ATTRIBUTE
DESCRIPTIONS BY HITTING RETURN

```
---------------------------------------------------
|             DATA DICTIONARY INDEX               |
|                                                 |
|    SSN        LAST NAME          FIRST NAME     |
|    ADDRESS    CITY AND STATE     ZIP CODE       |
|                                                 |
|    ----------------------------------------     |
|    | USE ^U TO MOVE UP, ^D TO MOVE DOWN,   |    |
|    | ^L TO MOVE LEFT, ^R TO MOVE RIGHT,    |    |
|    | RETURN TO SELECT OPTION               |    |
|    ----------------------------------------     |
|_________________________________________________|
```

USER USES ^D, ^R TWICE AND SELECTS 'ZIP CODE'

Figure 1. Typical session from user's point of view

```
 _____________________________________________________
|                 ZIP CODE                            |
|                                                     |
|      ENTER ONLY THE 5 DIGIT ZIP CODE; 9             |
|      DIGIT ZIP CODES ARE NOT YET SUPPORTED          |
|      EXAMPLE:  77001                                |
|                                                     |
|      ______________________________________         |
|      | USE RETURN TO ACCESS MAIN MENU     |         |
|      |____________________________________|         |
|                                                     |
|_____________________________________________________|
```

RETURN GETS USER BACK TO MAIN MENU FROM
WHERE FURTHER HELP MAY BE OBTAINED OR
THE USER MAY EXIT BACK TO THE PROGRAM

Figure 1, continued.

Thus, the program had to be smart enough to know where the call was issued from, call the appropriate menu/manual, respond to the user's inputs and then return the user to the point from which the help call was made. We also decided that, in order to improve the user friendliness aspect of the program, the user should have little or no typing to do - i.e., the program should be able to respond to special purpose function key inputs.

Of course, the use of the help manual itself must be self-descriptive in order to relieve the user from having to turn to a manual on the on-line help manual.

Figure 1 shows what a typical session, from the end-users point of view, might look like.

3. Implementation Particulars

3.1 The Text File

The text file created by the implementor of a particular help facility has certain restrictions and rules. Figure 2 illustrates what the text file should "look" like.

The name of the help file corresponds to the name by which the help file is identified within the program. This name is not seen by the user, only by the program. This name should always be the first line of the text file.

Following the name of the help file comes the "name" of the introduction - i.e., a name with which or by which the user can identify the help package accessed. This name also appears on a line by itself. Following the introduction name is the textual description of the introduction itself.

Following the introduction, the name of each menu selection which will be made available to the user, along with the explanation which will be provided if the user selects that option, appears. Each name (including the introduction name) appears on a separate line and each description or explanation is terminated with a # terminator, also on a line of its own.

The current limits imposed on the names and descriptions in the text file are:

- the names are limited to 15 character in length,
- the textual descriptions for each menu option and for the introduction are limited to 24 lines of 80 characters each, and
- 65 such descriptions (including the introduction description) can exist.

These limits are, of course, program constants that can be changed, as necessary, to meet the requirements at hand.

3.2 The help file package

Once the text file has been created, the creator can incorporate his or her help facility into the general help package by calling a program called PROTOTYPE. The basic task of PROTOTYPE is to create a binary file containing the information of the text file.

When PROTOTYPE is called, the contents of a binary file called

```
Payroll/personel
Payroll system
This panel allows input of new or update of existing
personell records regarding payroll
#
SSN
The employee's social security number, in the following
format: 111 22 3333
#
Last name
Employee's last name, up to 20 characters in length.
Upper and/or lowercase letters may be used.
#
First name
Employee's first name, up to 20 character's in length.
Upper and/or lower case letters may be used.
Example: Employee's name is Marie Elizabeth Ogden;
         enter Marie as first name, even if employee goes
         by a different name.
#
```

Figure 2. Partial contents of a typical text file containing information to produce an on-line help manual

INDEX.BIN are loaded into memory. This binary file is strictly a set of names and associated indices (or read/write head positions). Figure 3 illustrates the contents of INDEX.BIN.

PROTOTYPE first reads the name of the help package from the text file. This name is stored in the next available position in an array of records which contains help panel names and pointers to their arrays of menu selection options, described below. The first position in the array contains a count of the total number of help files available; this number must be updated each time a new help facility is added to the system. The indices or pointers associated with each name are actually read/write head positions into INDEX.BIN itself where the set of menu selections are listed in an array associated with that particular help package.

After the help package name, in the text file, the introduction and menu selection names, along with their descriptions, are found. PROTOTYPE reads the name of the introduction or menu selection option and stores the name in the next available position in an array of menu selection names for that particular help facility. Once again, the first element of each of these arrays contains a count of the number of menu selection options (including the introduction) available through the package. (The actual position of the introduction name is always the second array position since it is always the first description to follow the help package name in the text file.) PROTOTYPE then reads the textual description associated with the last name read and when the terminator (#) is encountered, it performs a DIRECT_IO write of the description into a file called DIRECT.BIN. The read/write head position of the write is stored in INDEX.BIN along with the menu selection/introduction name with which it is associated.

When EOF is encountered in the text file, and no violations have occurred, PROTOTYPE performs a DIRECT_IO write of the updated version of INDEX.BIN. Any violation of syntax rules encountered in the text file during the above process causes an abort, with no updating of the INDEX.BIN binary file.

The second direct_io or binary file is the DIRECT.BIN mentioned above. It simply contains the textual descriptions, in binary form, which the help package will use.

Figure 4 illustrates the relationship between INDEX.BIN and DIRECT.BIN.

4. Discussion

When the user request on-line help he or she first gets a general menu where one of three general options can be selected. These are:

1- a set of generalized commands/keys which pertain to all help packages - e.g., how to save his/her work, how to exit to the system without saving the work, etc.

2- The introductory section to the help facility which contains a general description of the help package itself and of the program/package with which it is associated.

3- A listing of the menu selection options available to the user.

```
 ________________________________________________________          _________________
|# of help    | index of      | index of       |                  | index of       |
| facilities  | option list   | option list    |                  | option list    |
|_____________|_______________|________________|       ...        |________________|
|             |  help pkg     | help pkg       |                  | help pkg       |
|             |  name         | name           |                  | name           |
|_____________|_______________|________________|                  |________________|
                                up to 501 such records

 ________________________________________________________         ________________
| # of          |  index into |  index into   |                 | index into |
| options       | DIRECT.BIN  | DIRECT.BIN    |                 | DIRECT.BIN |
|_______________|_____________|_______________|      ...        |____________|
|               | name of     | name of       |                 |  name of   |
|               |  option     | option        |                 |   option   |
|_______________|_____________|_______________|                 |____________|
                                  up to 64 records/array
                                  up to 501 such arrays (1 per help
                                     facility)
```

Figure 3. The basic structure of the direct_io
           file INDEX.BIN

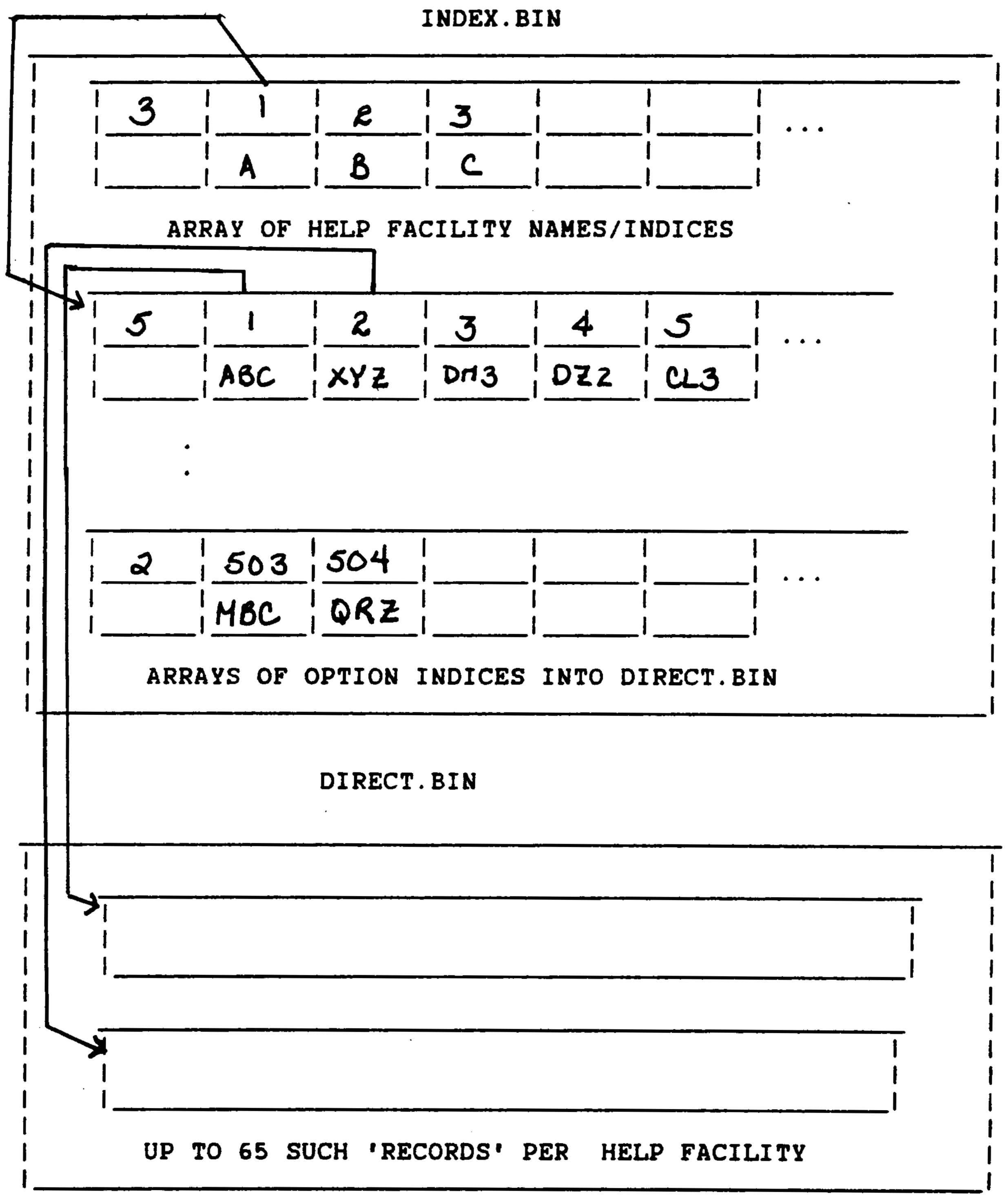

INDEX.BIN
3
1
2
3
A
B
C
...
ARRAY OF HELP FACILITY NAMES/INDICES
5
1
2
3
4
5
ABC
XYZ
DM3
DZZ
CL3
...
2
503
504
MBC
QRZ
...
ARRAYS OF OPTION INDICES INTO DIRECT.BIN
DIRECT.BIN
UP TO 65 SUCH 'RECORDS' PER HELP FACILITY

Figure 4. Relationship between INDEX.BIN and DIRECT.BIN

Each screen produced by a user selection contains a general set of directions for cursor movement, returning to previous screens, exiting, etc.

When the user selects option 1 above, PROTOTYPE is by-passed and the ON-LINE-HELP package calls a panel generation package to produce the "general help" screen, since the contents of this screen are the same regardless of where it was called from.

Options 2 and 3, on the other hand, use the information created by PROTOTYPE discussed above. When either of these is selected, INDEX.BIN is opened and the name of the help package associated with the program the user is in is searched for. When located, the pointers into DIRECT.BIN are made available through the array of pointers associated with that name. If option 2 was selected, the introduction section pointer (which, as mentioned earlier, is always the second record of the particular array) is used as the read/write head position into DIRECT.BIN where the textual description can be found.

If option 3 was selected by the user, a listing of menu options available is produced on the screen. This listing is available directly from INDEX.BIN since the names of the menu options (along with their indices into DIRECT.BIN) are stored therein. Since the menu selection items are listed in the order in which they appear in the INDEX.BIN array for that package, keeping track of the cursor movement, through simple addition and subtraction, also keeps track of the index into the array DIRECT.BIN for the element pointed to by the cursor. Thus, when the user does select a menu option, the index into DIRECT.BIN is directly available and the requested information can be displayed.

Figures 5 and 6 illustrate actual screens which the user might encounter; these are self-descriptive.

When the user exits the help mode, he or she is returned to the point from which the help call was made. This is made possible by saving the user's last used 'screen' in a buffer whose contents are re-displayed when the help facility is exited.

5. Summary and conclusion

As can be deduced from the above discussions, the idea behind the help facility is relatively simple. It is made unique by the fact that it is written in Ada and uses aspects of the language which make information retrieval rapid and simple. Specifically, the DIRECT_IO facility allows for random access into the help files. It is unnecessary to discuss the advantages of random access over sequential access.

The mere fact that the program is written in Ada implies a saving in terms of lines of code. This introduces the possiblity of eventually adapting the program to run at the micro-computer level, a major consideration in this day and age.

Additionally, since the program uses only standard Ada generics, it is portable to other systems. This is another aspect which must always be taken into consideration in writting any software package in the modern day world of computer programming.

```
 ____________________________________________________________________
|                                                                    |
|          ________________________________________________          |
|         | COMMANDS ON PANEL                              |          |
|         | PANEL PREFACE => BRIEF INTRO TO CURRENT        |          |
|         |                  PANEL                         |          |
|         | DATA DICTIONARY                                |          |
|         |________________________________________________|          |
|                                                                    |
|          _______________________________________________           |
|         | RETURN TO ACTIVE COMMAND MENU                 |           |
|         |_______________________________________________|           |
|                                                                    |
|          ______________________________________________            |
|         |  USE ^U TO MOVE UP, ^D TO MOVE DOWN,         |            |
|         |  RETURN TO SELECT OPTION                     |            |
|         |______________________________________________|            |
|____________________________________________________________________|

 ____________________________________________________________________
|                      PANEL COMMAND DEFINITION                      |
|                                                                    |
|     SAVE => SAVES ALL INPUT TO DATABASE                            |
|     EXIT                                                           |
|     LOAD                                                           |
|     ERASE                                                          |
|     HELP                                                           |
|                                                                    |
|                                                                    |
|      ______________________________________________________        |
|     |  USE ^U TO MOVE UP, ^D TO MOVE DOWN, RETURN TO       |        |
|     |  SELECT OPTION                                       |        |
|     |______________________________________________________|        |
|____________________________________________________________________|
```

Figure 5.

```
 ________________________________________________________________
|                                                                |
|                      PANEL PREFACE                             |
|                                                                |
|        ______________________________________________         |
|       | USE RETURN TO ACCESS COMMAND MENU             |        |
|       |_______________________________________________|        |
|                                                                |
|       THIS PANEL ALLOWS FOR INPUT OF EMPLOYEE                  |
|       PAYROLL RELATED INFORMATION.  INFORMATION                |
|       CURRENTLY HELD ON A PARTICULAR EMPLOYEE                  |
|       MAY BE ALTERED OR UPDATED VIA THIS PANEL.                |
|                                                                |
|________________________________________________________________|
```

Figure 6.

*Graduate students at The University of Houston--Clear Lake, Houston, Texas

SESSION F.5	REUSABILITY PANEL
Panel Chair:	Delores S. Moorehead Intermetrics Houston, Texas
Panel Members:	

Ron McCain
IBM Federal Systems Division
Houston, Texas

Ed Berard
EVB Software Engineering, Inc
Rockville, Maryland

Daniel McNichol
McDonnell-Douglas Astronautics Co.
St. Louis, Missouri

Rick Blumberg
Planning Research Corp.
McLean, Virginia

Norm Nise
California State Polytechnic University
and
Rockwell International
Downey, California

Elizabeth Wald
Naval Research Laboratory/STARS

SESSION F.6 DISTRIBUTED Ada PANEL

Panel Chair:

Roger Racine
The Charles Stark Draper Laboratory
Cambridge, Massachusetts

Panel Members:

Mike Kamrad
Honeywell Systems and Research Center
Minneapolis, Minnesota

Pat Rogers (Charles W. McKay)
University of Houston_Clear Lake
Houston, Texas

Trevor Mudge
Department of Electrical Engineering
University of Michigan
Ann Arbor, Michigan

Application and Systems Software in Ada: Development Experiences

Jim Kuschill
Computer Representatives, Inc.
Santa Clara, California

This presentation focuses on two issues: why CRI chose to convert its existing commercial software products to Ada and the the technical challenges we faced both before and during the rewrite process. The presentation will cover the following:

I. Environment
   A. Began the rewrite of software written in SPL and FORTRAN to Ada in 1983.
      1. Software included: relational DBMS, 4GL tools, and project management system.

II. Why Ada?
   A. Current and future maintenance considerations.
   B. Transportability had tremendous marketing advantages.

III. Planning Challenges
   A. Shortage of available programmers.
   B. Learning curve amongst own personnel.
   C. Unknown degree of diffulculty in the use of Ada for the development of application software.

IV. Technical Challenges
   A. Strong typing requirements of Ada affected the data conversions necessary for relational accessing.
   B. Ada packaging functions forced some new coding and routines to be written for an already mature product.
   C. Overloading capability smoothed the transition between some functions.

V. Opinions and Results
   A. The re-write process totaling approximately 250,000 lines of code is now in alpha test (will be in beta by the time of the SIGAda Conference).
   B. The learning curve was shorted than expected and differed by the nature of the language each programmer was accustomed to using previously.
   C. Maintenance problems and costs, as demonstrated during development will be vastly reduced as a result of Ada.
   D. The structure of Ada forces the writing of better routines, therefore better software.
   E. The time between a successfully compiled program and a completed program is drastically reduced because of Ada strict coding requirements.

N89-16370

Software Development: The PRODOC Environment and Associated Methodology

Joseph M. Scandura, Ph. D.
University of Pennsylvania

In its most basic sense software development involves describing the tasks to be solved -- including the given objects and the operations to be performed on those objects. Moreover, such descriptions must be precise in order for a computer (or human) to perform as desired. Unfortunately, the way people describe objects and operations typically bears little resemblance to source code in most contemporary computer languages.

There are two potential ways around this problem. One is to allow users to describe what they want the computer to do in everyday, typically imprecise English (or to choose from a necessarily limited menu of choices). This approach has some obvious advantages and a considerable amount of research is underway in the area. The approach, however, also has some very significant limitations: (a) it currently is impossible to deal with unrestricted English, and this situation is unlikely to change in the foreseeable future; and (b) even if the foregoing limitation is eventually overcome, the approach would still require the addition of complex, memory intensive "front ends". These "front ends" interact with the user's typically imprecise English statements and effectively "try to figure out" what the user intends. The result invariably is a system which is both sluggish in performance and limited in applicability.

The PRODOC methodology and software development environment is based on a second, we believe sounder, more flexible and possibly even easier to use approach. Rather than "hiding" program structure, PRODOC represents such structure graphically using visual programming techniques. In addition, the program terminology used in PRODOC may be customized so as to match the way human experts in any given application area naturally describe the relevant data and operations. This customized terminology is all based on a uniform, very simple syntax that might easily be learned by an intelligent human (in a few minutes time). The approach taken with PRODOC is general, as well as efficient and easy to use.

PRODOC employs a unique graphically supported approach to software development, and supports the entire systems software development process, from requirements definition and system design to prototyping, code generation and maintenance. Although radically different at a superficial level, PRODOC draws generally on our extensive research in structural learning (the science of cognitive, instructional and intelligent systems engineering, Scandura, 1986). It represents a major step in the direction of automating the process of Structural (cognitive task) Analysis (e.g., Scandura, Durnin & Wulfeck, 1974; Scandura, 1977, 1982, 1984a, 1984b). More specifically, a special rule construct (not to be confused with production rules) plays a particularly central role in PRODOC.

C-6

In the next section, we define more precisely what we mean by a rule and show how rules can be represented as Scandura FLOWforms. Next, we describe the PRODOC system itself. Finally, we provide an overview of the IMS System Development Methodology using PRODOC.

THE RULE CONSTRUCT

Rules have three major components: a domain or set of data structures on which the rule operates, a range or set of structures which the rule purports to generate and a procedure (e.g., Scandura, 1970). Rules have been shown to provide a convenient way to represent a wide variety of human cognitive processes as well as arbitrary computer systems (e.g., Heller & Reif, 1984; Scandura, 1969, 1971, 1973, 1977; Scandura & Scandura, 1980).

The term "rule" corresponds directly to the concept of a program. The "procedure" component of a rule (i.e., step-by-step prescriptions for carrying out the rule) corresponds directly to the procedural portion of a program. "Domain" and "Range" components of rules define problem schemes (i.e., classes of problems) and refer to input, output and intermediate (local) structures. Collectively, they correspond to the data structures on which programs operate. These correspondences are summarized below:

```
            Program                                 Rule
              /  \                                 /     \
             /     \                             /         \
           /         \                         /             \
       Data        Procedure                 /                 \
    Structures                    Domain/Range              Procedure
                              (input/output) Structures
```

In general, the execution of rule procedures involves both testing conditions and carrying out operations. Where the internal structure of a rule procedure is unimportant, the rule is "atomic" or elementary -- i.e., is viewed as nondivisible for present purposes. Those familiar with production rules will note that PRODOC rules are more general. The procedures of production rules consist solely of operations and, consequently, correspond to "atomic" rules.

In programming parlance, atomic rules correspond to program "subroutines." These include PRODOC "library rules". The extended version of PRODOC makes it possible to create libraries of such rules. These libraries make it easy for nonprogrammers (as well as programmers) to construct executable PRODOC rules.

As mentioned above, rules may be written in a language which is either understandable to humans and/or interpretable by computer. In either case, however, the same basic form of representation may be used. FLOWforms are easily understood by most people and can be used to represent arbitrary procedures (whether rule procedures or program procedures).

Like all structured procedures, FLOWforms may be refined arbitrarily. They are used for two purposes, one to represent procedures and two, to represent input/output data structures.

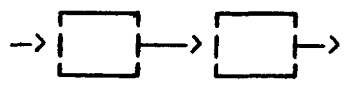
Roughly speaking, a procedure or algorithm is a recipe, process, technique, or systematic method for doing something. (The term "algorithm" is often preferred in computer science.) More precisely, according to Knuth (1968), a procedure or algorithm must:

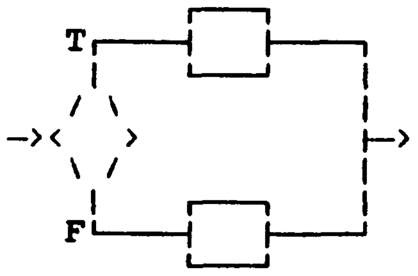
(1) always terminate after a finite number of steps,
(2) include only definite steps that are precisely defined, with actions that can be carried out rigorously and unambiguously,
(3) have an associated (possibly empty) class of inputs, or domain,
(4) generate at least one output, and
(5) be effective in the sense that all of the operations to be performed must be sufficiently basic that, in principle, they can be done exactly and in finite time by a person using pencil and paper.

Not all procedures are structured, however. Structured procedures are composed of substructures (components) or elements which have unique points of entry and exit. In order to insure this property, each step in a structured procedure must be decomposable into one of three basic types of components;

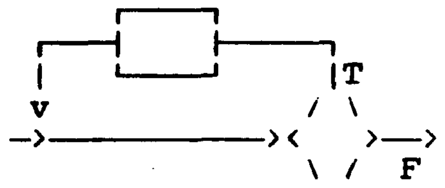
(a) Sequence of steps or operations,
(b) Conditional steps or branching (selection) and
(c) Iteration or looping.

These types are illustrated below both in terms of traditional flowcharts and Scandura FLOWforms. In the former case (a) the rectangles represent arbitrary operations (e.g., add a and b) and the diamonds represent (b) arbitrary selection or "if" conditions (e.g., If the building is over 20' tall, then...) and (c) arbitrary looping ("while") conditions (e.g., While there is still further to go...).


(a)

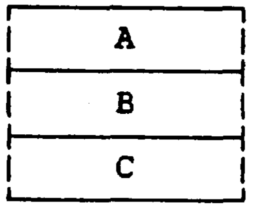


(b)

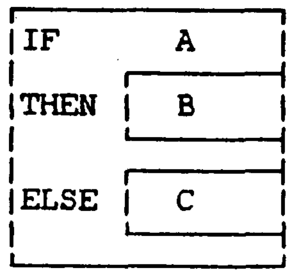


(c)

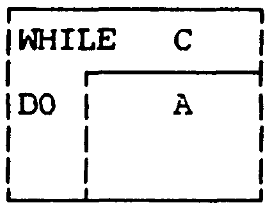
In Scandura FLOWform these three types of components are represented as shown below.



Sequence



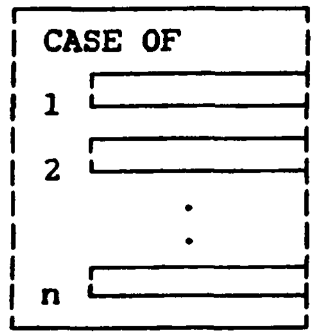
Selection (IF..THEN..ELSE)



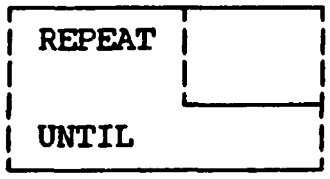
Iteration (WHILE..DO)

These three basic types of decomposition are univerally applicable and independent of any particular programming language (or any natural language for that matter). Moreover, used in combination via successive refinement, they have been proven adequate for any system design or programming task. Hence, there is no loss of generality in requiring that a procedure be structured.

Nonethless, it is often convenient to allow certain variations on the above. Some common variations on selections and iterations are shown below.

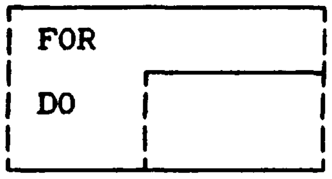


Selection (CASE)



Iteration

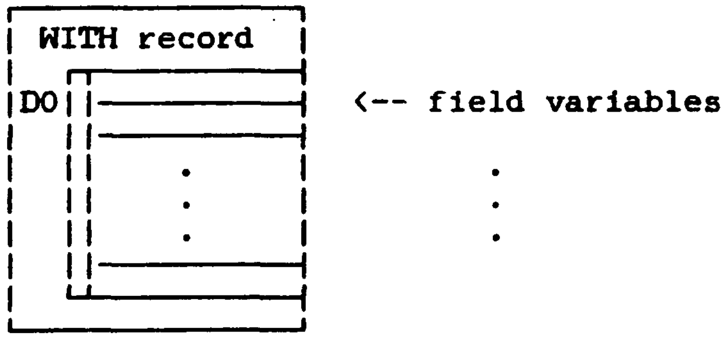
(REPEAT...UNTIL)



Iteration

(FOR...DO)

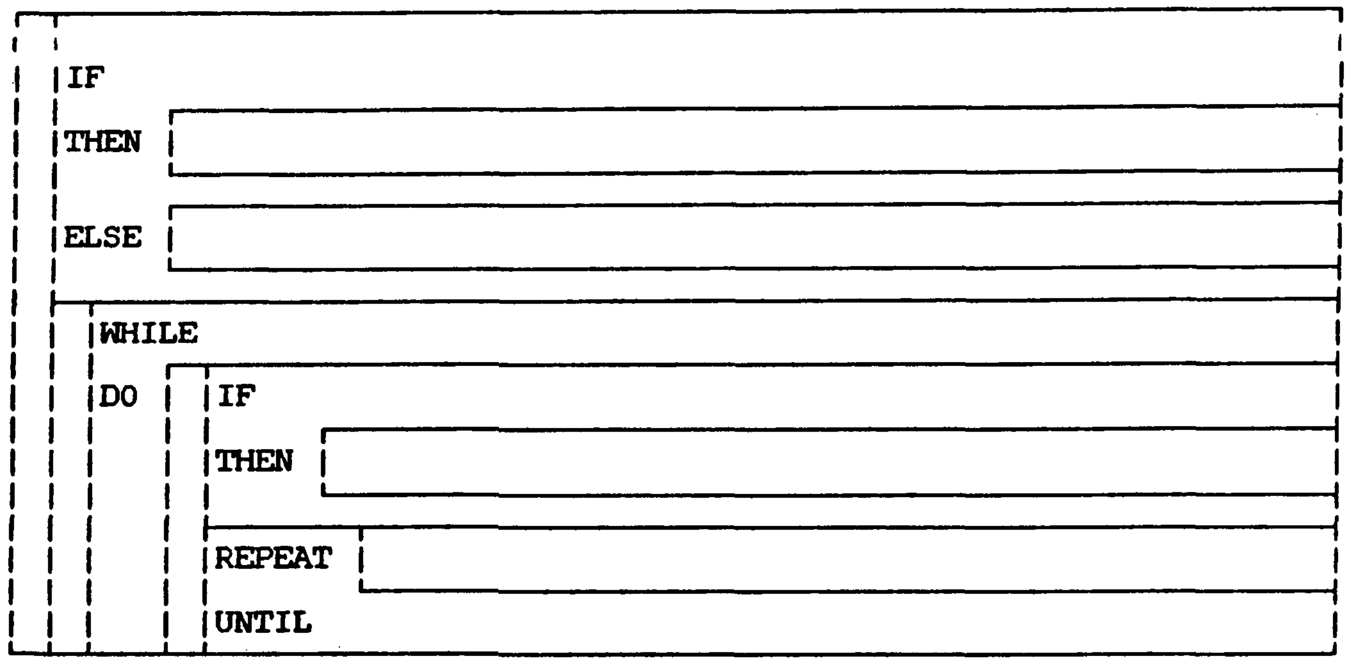
Although it does not fall into one of the three basic classes, Pascal also supports a WITH (Record..Do) structure. This is represented in FLOWforms as:



with (Pascal only)

In Scandura FLOWforms, sequence structures are often displayed using PRODOC with indentation to show level of refinement. This makes it easier to move about and otherwise manipulate FLOWforms on the screen. A sample FLOWform showing such indentation along with a variety of structure (decomposition) types follows:

[SAMPLE_R]:sample_FLOWform_structures   Copyright 1986 Scandura



Commands:Move keys,1..9,f,a,b,r,Del,t,m,d,c,e,s,^,z,g,l,w,?,F1,Esc

Parenthetically it is worth noting that FLOWform procedures may be recursive as long as the language in question supports recursion. This is certainly the case, for example, with Pascal, C, Ada and Lisp. This is not the case, however, with high level library rules (see next section) used in conjunction with PRODOC. To help insure future generalizability of the PRODOC system, library rules fully reflect all of the constraints imposed on the rule construct as defined in the structural learning theory (e.g., Scandura, 1977, 1981). In that theory, the role of recursion is handled exclusively in terms of higher order rules (which may operate on other rules) and an universal control mechanism. Recursion is not allowed in individual rules. This restriction has been shown to have important implications for diagnostic testing and learning (e.g., Scandura, 1980.)

Scandura FLOWforms also are used to represent rule domain (input) and range (output) structures. In general, domain and range structures may be characterized mathematically as partial orderings. The various components/elements may be viewed as ordered sets whose elements in turn may be ordered sets.

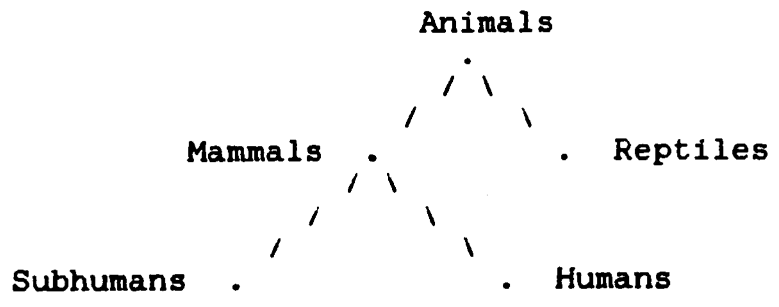
```
                    Domain A
                       /\
                     /    \
                   B        C
                 / | \     /|
               /   |   \  / |
             /     |     \  |
           E       |    /  \|
                   F   G    H
```

In the structure below, set A has elements B and C; B has elements E, F and H; C has G and H. Although element H appears twice in this FLOWform, it is simply a different display of the same element (something you will see when you edit one of them).

Although this representation looks similar to the CASE structure, the similarity is a bit deceptive. In procedures, CASE structures have both condition variables and operations. The condition occupies a distinguishing position to the right of the word "CASE" and may be thought of as the first CASE element.

```
[SAMPLE]:Sample_DOMAIN_FLOWform              Copyright 1986  Scandura

┌───────────────────────────────────────────────────────────────────┐
|                                                                   |
|[DOMAIN]:                                                          |
| ┌─────────────────────────────────────────────────────────────────┤
| |[A]:                                                             |
| | ┌───────────────────────────────────────────────────────────────┤
| | |[B]:                                                           |
| | | ┌─────────────────────────────────────────────────────────────┤
| | | |[E]:                                                         |
| | | └─────────────────────────────────────────────────────────────┤
| | | ┌─────────────────────────────────────────────────────────────┤
| | | |[F]:                                                         |
| | | └─────────────────────────────────────────────────────────────┤
| | | ┌─────────────────────────────────────────────────────────────┤
| | | |[H]:                                                         |
| | | └─────────────────────────────────────────────────────────────┤
| | └───────────────────────────────────────────────────────────────┤
| | ┌───────────────────────────────────────────────────────────────┤
| | |[C]:                                                           |
| | | ┌─────────────────────────────────────────────────────────────┤
| | | |[H]:                                                         |
| | | └─────────────────────────────────────────────────────────────┤
Commands:Move keys,1..9,f,a,b,r,Del,t,m,d,c,e,s,^,z,g,l,w,?,F1,Esc
```

Notice that this representation is not quite a tree since element H belongs to both sets B and C. Of course, partial orderings do include trees as a common subset. A simple example of a tree is given below.



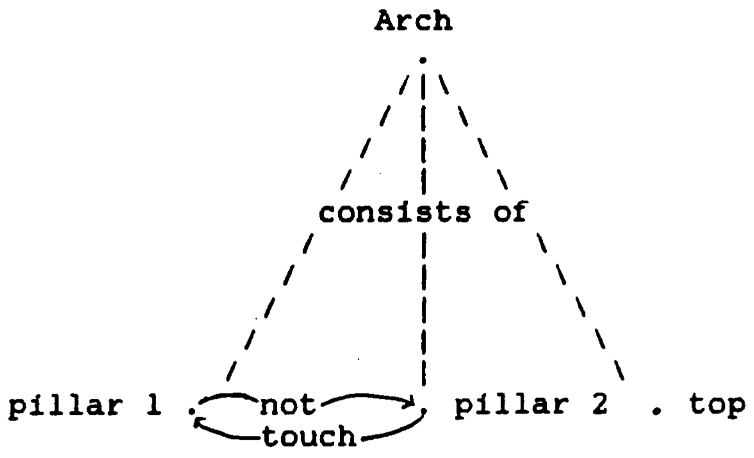
Since rule data structures are restricted to partial orderings it is true that FLOWforms cannot directly represent cyclical relationships. In the case of software development, however, this restriction is more apparent than real. Cyclic relationships can serve two quite different purposes:

(1) They can be used to summarize connections among nodes (e.g., computer terminals) in a complex system.

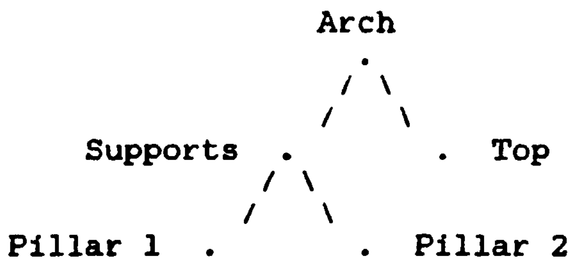
(2) They can be used to represent nonhierarchical data structures, where the relationships are not necessarily monotonic.

In the former case, for example, the connections typically represent a sharing of data represented by the nodes. Just as data at any given node can be operated on by resident programs, programs also are needed to transfer data from one node to another. Thus, the cyclic networks themselves correspond to sets of programs, each of which may be represented in terms of a rule FLOWform. Such networks, in effect, provide a convenient way to represent the overall high level structure of a system of programs but they say relatively little about software development per se.

The figure below illustrates the latter case -- data which a program procedure might operate on.



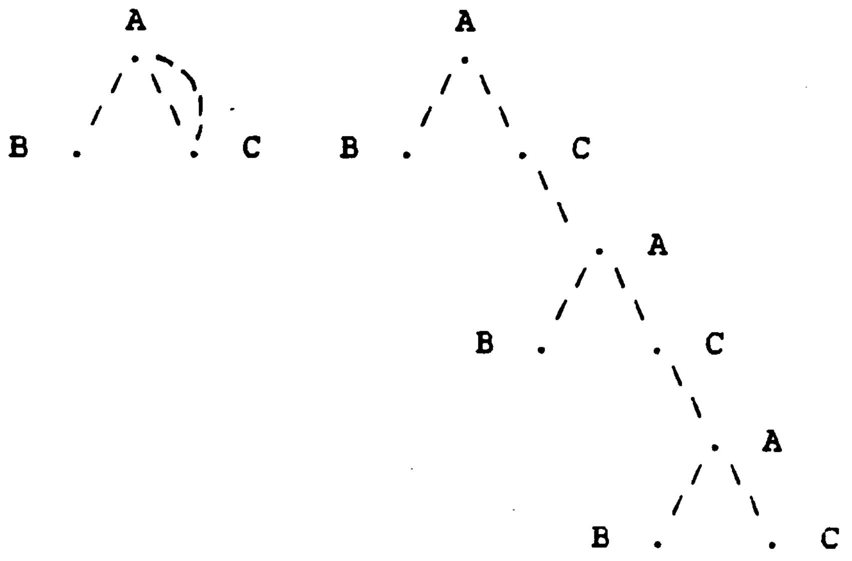
In this case, notice that the nodes "pillar 1" and "pillar 2" are superordinate to each other. This is not allowed in a partial ordering relationship. As with successive top-down structured refinement of procedures, most software engineers favor a hierarchical (partially ordered) approach to data structure design. Thus, for example, the above Arch structure might be represented hierarchically as



where the definition of "supports" may include "not touching". In fact, the latter figure seems more natural. Accordingly, arches consist of two types of entity: supports and tops. In turn, (at least) two supports are needed.

Nonetheless, it is fair to ask whether cyclic relationships are necessary for some purposes. While we do not know of any definitive answer to this question, it would appear that the answer is "no". Just as any procedure can be represented as a structured procedure, cyclic data structures can be represented in terms of partial orderings. To see this, notice that cycles correspond to infinite hierarchies (e.g, pillar 1 --> pillar 2 --> pillar 1 --> pillar 2 -->).

However, any given cycle can be realized only a finite number of times in the real world. Hence, cyclical relationships can be represented by finite successive refinement of the cycles in question. Consider, for example, the cyclic graph on the left (below) and the equivalent partial ordering on the right. While the cyclic graph looks simpler, it camouflages the fact that the cycle is repeated only twice.



In effect, the apparent loss of representational simplicity is at least partially overcome by the more precise characterization provided by the partial ordering. The suppression of such details is not appropriate in actual software development.

It would appear, just as one can always construct a structured procedure equivalent to given "spagetti" code, one can always construct a partially ordered data structure that is equivalent to any given cyclical data structure.

PRODOC

Using PRODOC, rule data structures and procedures are constructed in a top-down structured fashion and represented in terms of Scandura FLOWforms. As we have seen, FLOWforms look similar to Nassi-Shneiderman flow charts, but they make better use of the rectangular screen and allow simultaneous display of as many (or as few) levels of representation as may be desired.

A procedure FLOWform with several levels of refinement might be displayed by PRODOC as illustrated below. At the highest level, for example, data structures and procedures each consist of a single high level description (component). Various components, in turn, are decomposed into one or more lower level elements.

Insert FLOWform showing several levels

PRODOC consists of four distinct but complementary and fully compatible software productivity and quality assurance environments. Each of these environments (described below) makes use of Scandura FLOWforms.

Relationships among the first three PRODOC environments as well as the way they may be used in developing applications software is represented schematically on the following page.

Insert first schematic here
----

(1) Applications Prototyping Environment (with interpreter and expert assistant generator) (PRODOCea) - is suitable for use by nonprogrammers as well as programmers for designing, documenting, implementing, and maintaining software systems in an integrated, graphically supported, top-down structured environment. In addition to English text, the availability of greatly simplified, high level library rules makes PRODOCea ideal for rapid prototyping. The availability of graphical support for input and output data structures also makes it possible to directly reflect arbitrary semantic properties.

The current version of PRODOCea employs a fairly general but relatively low level set of library rules designed largely for testing purposes. The current library includes a variety of:

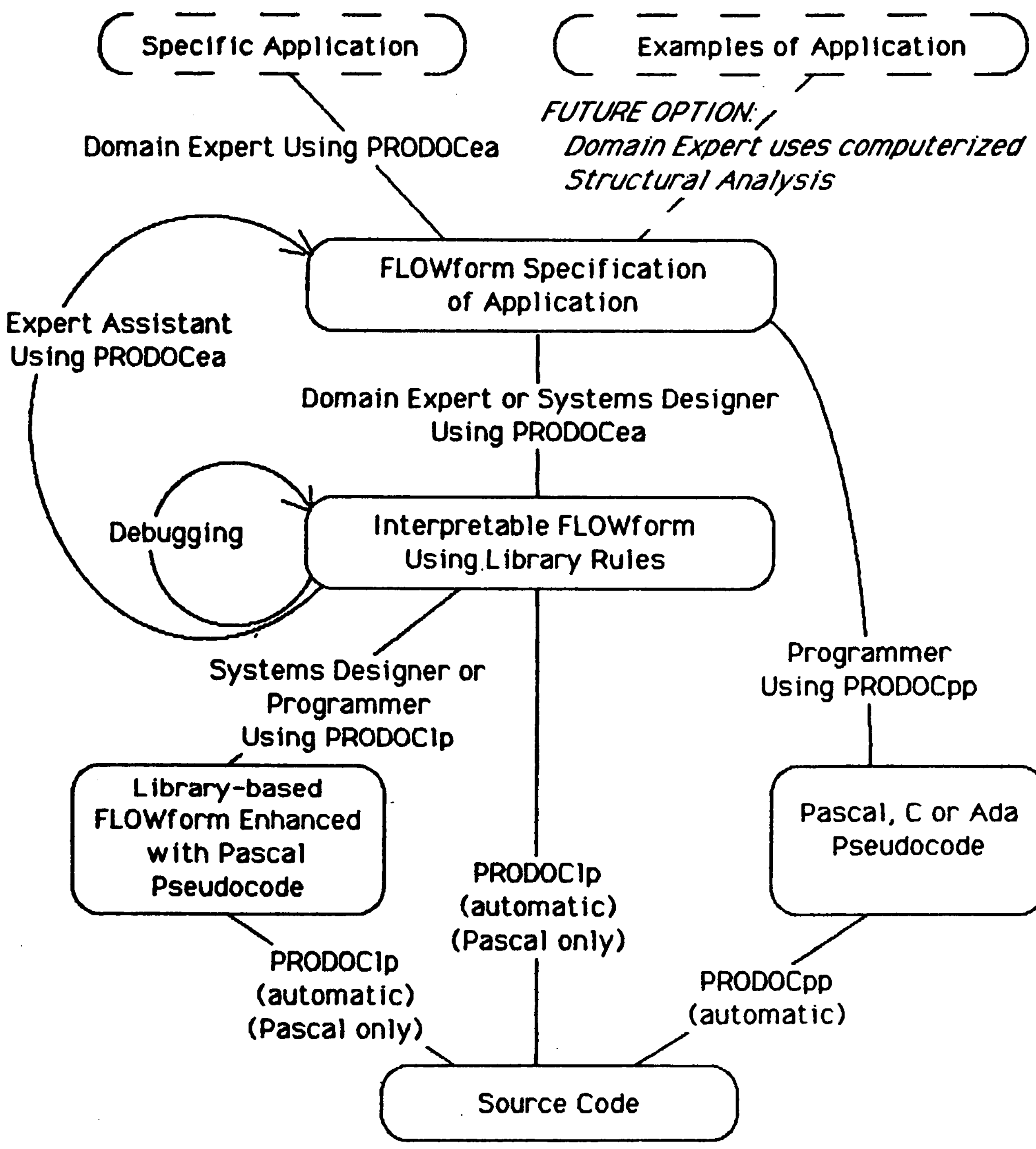
input/output operations [e.g., display (ELEMENT, 05-12-86

[SORT]:sort Copyright 1986 Scandura

Sort up to 500 numbers;print result

```
write ('How many numbers (1 to 500) to be sorted?  ')
readln (n)
. Prompt user, then get numbers.
    writeln ('Enter below numbers to be sorted. Press <Return> after each.')
    . Get the numbers from the user.
    FOR i:=1 to n
    DO  readln (a[i])
. Sort them.
FOR i:= 1 to n-1
DO  . Scan thru items and swap if necessary.
    FOR j:= 1 to n -i
    DO  . Compare and swap if necessary.
        IF a[j]> a[j+1]
        THEN  . Swap
                  temp:= a[j]
                  a[j]:= a[j+1]
                  a[j+1]:= temp
. Identify and then print the resulting ordered set.
    writeln
    writeln ('The resulting order is:')
    . Print the result.
    FOR i:= 1 to n
    DO  writeln (a[i]:2)
```

IMS's PRODOC
Software Development Environment:



Specific Application
Examples of Application
FUTURE OPTION:
Domain Expert uses computerized Structural Analysis
Domain Expert Using PRODOCea
FLOWform Specification of Application
Expert Assistant Using PRODOCea
Domain Expert or Systems Designer Using PRODOCea
Debugging
Interpretable FLOWform Using Library Rules
Systems Designer or Programmer Using PRODOClp
Programmer Using PRODOCpp
Library-based FLOWform Enhanced with Pascal Pseudocode
PRODOClp (automatic) (Pascal only)
Pascal, C or Ada Pseudocode
PRODOClp (automatic) (Pascal only)
PRODOCpp (automatic)
Source Code

DISPLAY_PARAMETERS), load (DOS_NAME, DRIVE, FILE_TYPE)],

other operations [(e.g., insert_component_after (VALUE, SET, PREVIOUS_COMPONENT), delete_component (SET, COMPONENT)],

functions [e.g., add (ADDEND 1, ADDEND 2), modulo (X, BASE), find (VALUE, SET)],

conditions [e.g., match (STRING 1, STRING 2), less_than (X,Y)],

logical connectives [e.g., and (EXPRESSION 1, EXPRESSION 2)],

and assignment (i.e., ELEMENT := VALUE).

The user also has the option of creating hierarchies of input/output data structures which directly reflect the reality they represent. Alternatively, inessential aspects of this structure may be suppressed. In this case, PRODOC automatically generates a formal equivalent of the needed data structures (i.e., declarations). Once "initialized" in this way, PRODOC library rules may be executed immediately in interpretive mode for purposes ranging from simple execution to debugging.

In conjunction with PRODOC's Library Generation facilities (see (4) below), custom versions of PRODOCea (and PRODOClp) can quickly be created to accommodate library rules to facilitate rapid prototyping in arbitrary semantic properties.

A unique feature of PRODOCea is its ability to immediately execute not only interpretable library rules but statements written in ordinary English. This makes it possible to actually run through a proposed system design before it has even been prototyped in terms of high level library routines, let alone reduced to standard program code. An additional advantage is that it makes the difficult and expensive process of developing many expert systems almost trivial. Once an (nonprogrammer) expert knows what a human/computer assistant is to do, it is a simple task to develop a computerized expert assistant or performance aid to assist less qualified personnel in performing the required tasks.

(2) Applications Prototyping Environment (for use with a Pascal compiler) (PRODOClp) - is identical to PRODOCea in so far as prototype design and the use of library rules in rapid prototyping is concerned. Instead of an interpreter, however, PRODOClp includes a much generalized code generator which makes it possible to arbitrarily mix Pascal code with library rules, thereby gaining the prototyping advantages of any number of customized, arbitrarily high level languages, along with the flexibility of Pascal. This feature makes it possible, for example, for a programmer to speed up or otherwise add finishing touches to a working prototype created by a nonprogrammer.

(3) Programming Productivity Environment (PRODOCpp) - has all of the design, etc. features of PRODOCea. PRODOCpp comes in standard form which supports source code in any programming language.

(Incidentally, PRODOC can be used as a full-function idea processor. This text, for example, was prepared using PRODOC exclusively.)

In addition, pseudo code support is available as an option for Pascal, C, Ada and other programming languages. For example, Pascal, C and Ada syntax and other routine aspects of code generation (e.g., BEGINS..ENDS, etc.) are all generated automatically. The result effectively combines the clarity and ease of use of high-level fourth generation languages with the flexibility of third generation languages. These options also include syntax checking, consistency checking and automatic declarations generation. Current plays call for adding pseudo code support for other third and fourth generation languages as needed.

A sample FLOWform for sorting numbers and the corresponding Pascal source code are shown on the next page.

Insert Sort FLOWform and Code

----

(4) Library Generator (PRODOClg) - makes it possible to integrate available rule libraries and new library rules into either PRODOC prototyping environment, thereby creating customized versions of PRODOC for particular families of applications. Since this requires access to PRODOC source code, customized versions of PRODOC will normally involve a collaborative effort involving our development team and software specialists in particular application areas.

The use of PRODOClg in developing customized versions of PRODOCea and PRODOClp is represented schematically on the next page.

Insert schematic for PRODOClg here

01-23-86

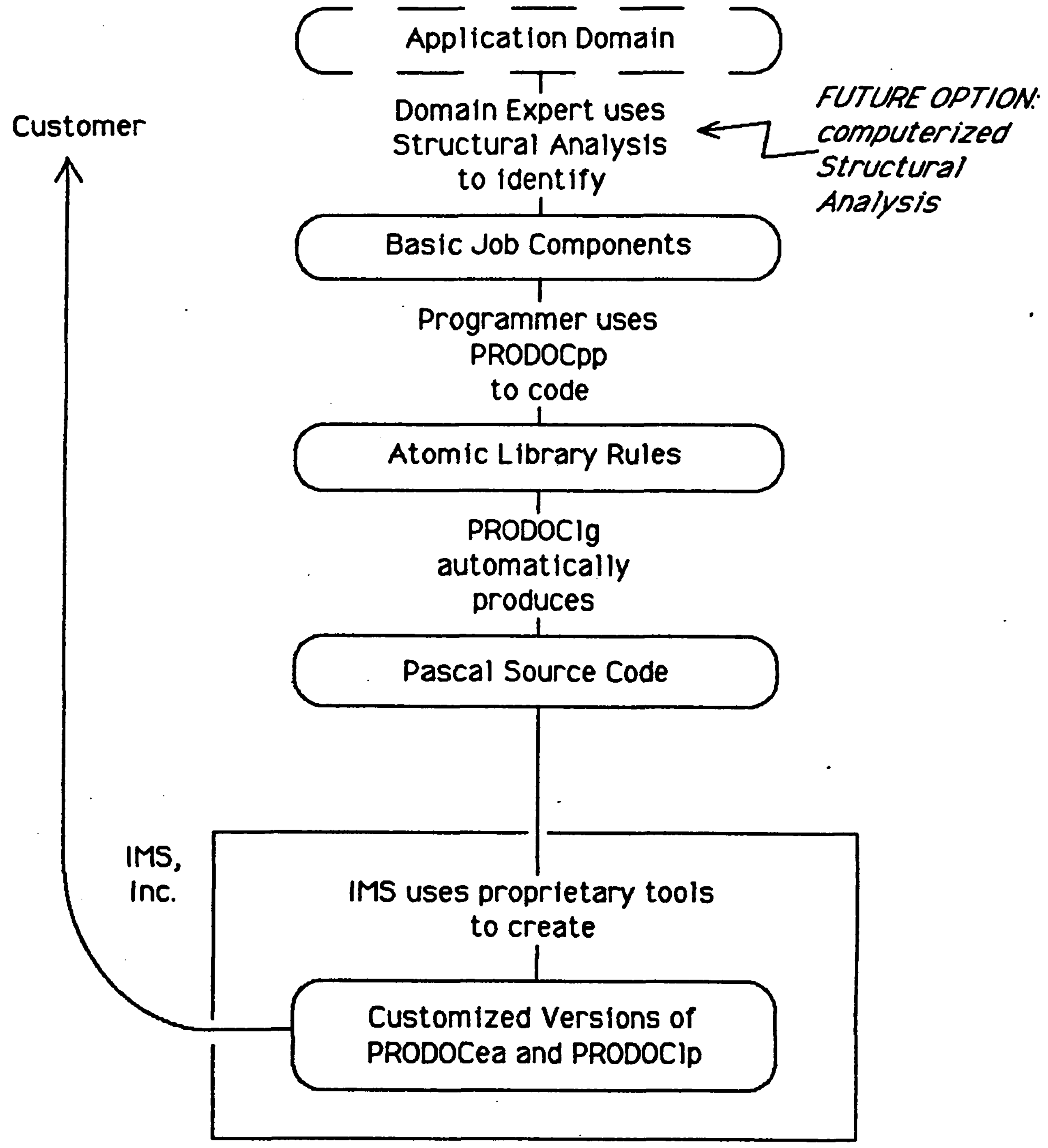
[SORT]:sort Copyright 1986 Scandura

Sort up to 500 numbers;print result

```
write ('How many numbers (1 to 500) to be sorted?  ')
readln (n)
    writeln ('Enter below numbers to be sorted. Press <Return> after each.')
    FOR i:=1 to n
    DO  readln (a[i])
FOR i:= 1 to n-1
DO  FOR j:= 1 to n -i
    DO  IF a[j]> a[j+1]
        THEN    temp:= a[j]
                a[j]:= a[j+1]
                a[j+1]:= temp
    writeln
    writeln ('The resulting order is:')
    FOR i:= 1 to n
    DO  writeln (a[i]:2)
```

```
PROGRAM sort;

VAR n : INTEGER;
    i : INTEGER;
    a : ARRAY[1..500] OF INTEGER;
    j : INTEGER;
    temp : INTEGER;

BEGIN
  { Sort up to 500 numbers;print result }
  BEGIN
    write ('How many numbers (1 to 500) to be sorted?  ');
    readln (n);
    { Prompt user, then get numbers. }
    BEGIN
      writeln ('Enter below numbers to be sorted. Press <Return> after each.');
      { Get the numbers from the user. }
      FOR i:=1 to n DO
        readln (a[i])
    END;
    { Sort them. }
    FOR i:= 1 to n-1 DO
      { Scan thru items and swap if necessary. }
      FOR j:= 1 to n -i DO
        { Compare and swap if necessary. }
        BEGIN
        IF a[j]> a[j+1] THEN
          { Swap }
          BEGIN
            temp:= a[j];
            a[j]:= a[j+1];
            a[j+1]:= temp
          END
        END;
    { Identify and then print the resulting ordered set. }
    BEGIN
      writeln;
      writeln ('The resulting order is:');
      { Print the result. }
      FOR i:= 1 to n DO
        writeln (a[i]:2)
    END
  END
END.
```

IMS's Structural Analysis Methodology and PRODOClg Library Generator:


Application Domain
Domain Expert uses Structural Analysis to identify
FUTURE OPTION: computerized Structural Analysis
Basic Job Components
Programmer uses PRODOCpp to code
Atomic Library Rules
PRODOClg automatically produces
Pascal Source Code
IMS uses proprietary tools to create
Customized Versions of PRODOCea and PRODOClp
IMS, Inc.
Customer

OVERVIEW OF THE SYSTEM DEVELOPMENT METHODOLOGY

Collectively, the various PRODOC environments provide a complete software development system, including requirements definition, systems design and documentation, prototype development, code generation and program maintenance. For this purpose, rules (represented in terms of data structure and procedure FLOWforms) provide an unique visual and uniform type of representation that can be used throughout.

The PRODOCea applications prototyping environment is designed primarily for use by system designers (in conjunction with intended users). (Given some initial training, in fact, it also can and has been used independently by end users.)

In this context, PRODOCea can be used in system analysis and requirements definition. System analyses will normally involve very high level descriptions of the various system states (data structures) and processes in ordinary English. Data FLOWforms will normally be used to describe the states, and transitions between states will be described at a high level in terms of procedure FLOWforms. Should the designer wish, these descriptions may include hardware, personnel and other development requirements.

During the requirements definition phase, users will develop more detailed descriptions of the key states and transitions. This is accomplished by successive refinement of the very high level system descriptions, all in an integrated environment.

PRODOCea makes it possible to "execute" these systems analyses and/or requirement definitions dynamically. That is, one can simulate transitions between various states of the to-be-developed system, thereby giving the user a better feeling for how the system might operate in practice.

As is well known, the distinction between requirements definition and program design is largely arbitrary and depends on one's perspective. In the former case, definition of the key states of the system, and of the transition procedures connecting them are described in largely functional, real world terms. Conversely, program designs typically are represented in terms of constructs associated with programming languages.

The various PRODOC prototyping environments are associated with given atomic rule libraries. Since rule libraries are designed to accommodate particular families of applications, both the data structures these rules operate on, as well as the rules themselves, directly reflect application realities.

Consequently, library rules (including both data structures and atomic rules) might be used directly in the case of requirements definition. Indeed, the resulting definitions might be interpreted directly (by PRODOCea) where the terminal (most refined) elements of the key transition procedures correspond to atomic rules in the associated library.

It may, in fact, still be possible to directly create an operational system even where the terminal elements of a systems definition or design are not already available as library rules. This might be accomplished in either of two ways:

(1) New library rules might be selected from available libraries and/or created (e.g., using PRODOCpp). These new rules can be integrated automatically to form a new library using PRODOClg. PRODOClg generates complete Pascal code which can, in turn, be linked with either PRODOC prototyping environment to create a custom version (of either). This new custom version, then, can be used to directly interpret the original systems definition or design (formulated in terms of atomic rules in the new library).

(2) The requirements definition stage might be further developed as normally is done into a detailed system design. In this case the data structures and procedures (represented in terms of applications reality) are reformulated in terms of data structures and operations more closely associated with some target source language. These more detailed designs, then, are converted to code using PRODOCpp. For this purpose, one can enter complete source code using PRODOCpp's default "text" files. Alternatively, in conjunction with available language-specific files, one can simply enter pseudo code. In the latter case, syntax and consistency checking and declarations and source code generation, may be performed automatically.

PRODOClp serves a supplemental role in the above context. For example, Pascal pseudo code can be used to supplement whatever library rules happen to be used in a given design. This can be done without restriction. Given the resulting library/Pascal pseudo code combination, PRODOClp can be used to generate complete Pascal source code ready for compilation.

PRODOClp also serves a useful role even where all elements of a design consist of library rules. Although the design can be interpreted, tested and debugged using PRODOCea, execution efficiency can usually be greatly improved via compilation. In this case, PRODOClp can be used to convert the given design (represented solely in terms of library rules and meaningful data structures) into complete Pascal source code ready for compilation.

Perhaps the single most important advantage in following the foregoing methodology is that of program maintenance. Given the integrated, fully interchangeable nature of the various PRODOC environments, there is no justifiable reason why system requirements or design, program documentation, or code should ever get out of synchronization. Consequently, finding one's way around in even very complex systems is several orders of magnitude easier than is normally the case. Furthermore, the printed documentation provides additional features that are especially useful with large system segments.

In developing smaller programs, of course, it may be possible to bypass some of the above steps. Thus, one has the choice of creating and simply using an applications prototype as is, or of designing and coding the program using PRODOCpp directly (e.g., in conjunction with particular sets of PRODOCpp pseudo code language support files).

At this point, it may be unclear how we propose to deal with the various other representational systems that are commonly used by designers. In this regard, we take essentially the same position that Martin and McClure (1985) take with respect to their "action diagrams": Although the methodologies may appear to differ, all of the commonly used forms of representation are either equivalent (to ours) or incomplete. In fact, while action diagrams are formally equivalent to procedure FLOWforms, we do not believe that they display overall structure nearly as clearly.

By way of summary, using PRODOC has the advantage of placing requirements definition, systems design, prototyping and program coding (not to mention system maintenance) on the same plane. System designs, prototypes, and program code are viewed within an integrated environment, which is far easier to understand, revise, debug, and modify than is normally the case. Put somewhat differently, developing and maintaining executable (interpretable or compilable) prototypes and/or source code is a natural extension of system design and documentation, and vice versa. In short, PRODOC supports the entire systems software management and development process, from requirements definition to code generation.

Those of us who have been involved in the creation of PRODOC are fond of pointing out that PRODOC has literally been indispensable in its own creation. Indeed, we would not even consider taking on a new programming task without using PRODOC.

REFERENCES

Heller, J. I. and Reif, F. Prescribing effective human problem-solving processes: Problem description in physics. Cognition and Instruction, 1, 177-216, 1984.

Knuth, D. E. The Art of Computer Programming Vol. 1: Fundamental Algorithmes. Reading, MA: Addison-Wesley, 1968.

Martin, J. and Mc Clure, C. Action Diagrams: Clearly Structured Program Design. Englewood Cliffs, NJ: Prentice-Hall, 1985.

Scandura, J. M. New directions for theory and research on rule learning: II. empirical research, Acta Psychologica, 1969, 29, 101-133.

Scandura, J. M. The role of rules in behavior: toward an operational definition of what (rule) is learned. Psychological Review, 1970, 77, 516-533.

Scandura, J. M. Deterministic theorizing in structural learning: three levels of empiricism. Journal of Structural Learning, 1971, 3, 21-53.

Scandura, J. M. Structural Learning I: Theory and Research. London/New York: Gordon and Breach Science Pub., Inc., 1973.

Scandura, J. M. Problem Solving: A Structural/Process Approach with Instructional Implications. New York: Academic Press, 1977.

Scandura, J. M. Theoretical foundations of instruction: a systems alternative to cognitive psychology. Journal of Structural Learning, 1980, 6, 347-394.

Scandura, J. M. Problem solving in schools and beyond: transitions from the naive to the neophyte to the master. Educational Psychologist, 1981, 16, 139-150.

Scandura, J. M. Structural (cognitive task) analysis: a method for analyzing content. Part I: background and empirical research. Journal of Structural Learning, 1982, 7, 101-114.

Scandura, J. M. Structural analysis. Part II: toward precision, objectivity and systematization. Journal of Structural Learning, 1984, 8, 1-28. (a)

Scandura, J. M. Structural analysis. Part III: validity and reliability. Journal of Structural Learning, 1984, 8, 173-193. (b)

Scandura, J. M. Structural learning: the science of cognitive, instructional and intelligent systems engineering. Journal of Structural Learning, 1985, 8, i-ii.

Scandura, J. M. PRODOC: The PROfessional self-DOCumenting programming productivity environment. Journal of Structural Learning, 1986, 9, 101-105.

Scandura, J. M., Durnin, J. H. and Wulfeck, W. H. Higher-order rule characterization of heuristics for compass and straight-edge constructions in geometry. Artificial Intelligence, 1974, 5, 149-183.

Scandura, J. M. and Scandura, A. B. Structural Learning and Concrete Operations: An Approach to Piagetian Conservation. New York: Praeger Sci. Publ., Inc. 1980.

N89-16371

A Database Management Capability for Ada*

Arvola Chan
Sy Danberg
Stephen Fox
Terry Landers
Anil Nori
John M. Smith

Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142

This project is supported jointly by the Advanced Research Projects Agency of the Department of Defense (DARPA) and the Naval Electronics Systems Command (NAVELEX) under contract N00039-82-C-0226. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NAVELEX, or the U.S. Government.

*Ada is a Registered Trademark of the U.S. Government (AJPO)

1. Introduction

The data requirements of mission-critical defense systems have been increasing dramatically. Command and control, intelligence, logistics, and even weapons systems are being required to integrate, process, and share ever increasing volumes of information. To meet this need, systems are now being specified that incorporate database management subsystems for handling storage and retrieval of information. Indeed, it is expected that a large number of the next generation of mission-critical systems will contain embedded database management systems. Since the use of Ada has been mandated for most of these systems, it is important to address the issues of providing database management capabilities that can be closely coupled with Ada.

Under sponsorship by the Naval Electronics Systems Command and the Defense Advanced Research Projects Agency, Computer Corporation of America has been investigating these issues in the context of a comprehensive distributed database management project. The key deliverables of this project are three closely related prototype systems implemented in Ada.

1. LDM (local data manager): an advanced, centralized database management system that supports a semantically rich data model designed to improve user productivity. It can be used either stand alone or as an integral part of the other two prototype systems.
2. DDM (distributed data manager): a homogeneous distributed database management system built on top of a collection of LDMs in a computer network. It supports the transparent distribution and replication of data in order to provide efficient access and high availability.
3. Multibase: a retrieval-only system that provides a uniform interface through a single query language and database schema to data in preexisting, heterogeneous, distributed databases. It utilizes LDM for managing its local workspace during the processing of a global query.

All three systems are designed to support identical interfaces for interactive use and for use through application programs written in Ada. Fundamentally, they support a "semantic" data model that captures more application semantics than conventional data models. The interactive language is called Daplex. Daplex has been designed to be an Ada compatible database sublanguage. The syntax of many of its constructs for data definition and data manipulation has been borrowed from Ada. The application programming interface is called Adaplex. It consists of an expression-level integration of Daplex's data manipulation constructs with Ada. This paper identifies a set of requirements for a modern database management capability for Ada that has driven our design for the aforementioned prototype systems. It provides an overview of the Daplex and Adaplex languages, and a summary of the functional capabilities and technical innovations we have incorporated in the LDM, DDM, and Multibase systems.

2. Requirements

Providing a database management capability for Ada is not an easy task. Our goal is to provide a complete set of modern database management capabilities which are consistent with the style and philosphy of Ada and which are well integrated with the Ada language and its support environments. This section summarizes the major requirements of a database management capability for Ada. These requirements can be grouped into three general areas: classes of databases that must be supported, operating environments, and compatibility with Ada.

Classes of databases

Ada programs will need to access three classes of databases. The first class consists of centralized databases. These databases reside at a single location and are managed by a DBMS that executes on a single computer. The second class consists of distributed databases. These databases can be fragmented, distributed, and replicated across a number of (possibly geographically separated) sites. They are managed by a DBMS that executes on a number of computers that are connected by a communications network. Distributed databases provide improvements in reliability, survivability, and expandability over centralized databases. The third class is pre-existing databases. These are databases (possibly centralized or distributed) that are managed by existing DBMSs. These DBMSs are not implemented in Ada. They provide different sets of functional capabilities and support different interface languages. An important requirement for an Ada database capability is to provide a single Ada interface to all of the above classes of databases. In other words, the particular class of database being accessed should be transparent to the Ada database application programmer.

Operating Environments

An Ada DBMS must be able to operate effectively in both an Ada programming support environment (APSE) to facilitate the development of Ada database application programs, and in an Ada run time environment to support the execution of these programs. To provide for the needs of these two environments, the DBMS must have two operating modes: shared and embedded. Shared mode is normally used in an APSE. A single copy of the DBMS supports the simultaneous development of multiple Ada database application programs in this mode. The interface between the application programs and the DBMS is a loosely-coupled one, each being executed as a separate Ada program. Thus, each application program can be changed without impacting the DBMS or other application programs. Embedded mode is typically used in a run time environment. Once the application programs have stabilized, they can be loaded together with the DBMS into a single Ada program. The applications and the DBMS then operate as separate Ada tasks that synchronize and communicate via rendezvous, thereby achieving a higher degree of interface efficiency at the expense of reduced flexibility. Embedded mode is less flexible than shared since a change to one application causes the other application and the DBMS to be relinked.

Compatibility with Ada

Ada has made a large contribution to improving program integrity through strong type checking at compile time and constraint checking at run time. It is important that an Ada DBMS provides the same degree of integrity on the Ada program data that it manages. An Ada DBMS should support all of the Ada data types, including derived types, subtypes, and type attributes. It should also support the same degree of run time constraint checking. Note that this cannot be easily (or efficiently) accomplished by simply providing an Ada interface to an existing (non-Ada) DBMS. Let us illustrate this with a simple example. Suppose an Ada programmer wants to store a set of employee records in a database. The Ada type definitions for this record may look like:

```
type YEARS is new INTEGER range 0..50;'

type EMPLOYEE is
  record
    NAME              : STRING(1..30);
    YEARS_OF_SERVICE  : YEARS;
    SALARY            : INTEGER;
  end record;
```

Suppose that the Ada programmer writes a program that contains a transaction that adds one to the YEARS_OF_SERVICE component of each employee record. There are two ways to process this transaction. One way is to retrieve the YEARS_OF_SERVICE component for each record in the database and return it to the application program, add one and then store it back in the database. This is a very inefficient way of processing since it results in a lot of data being sent from the DBMS to the application program and then back again. A much more efficient method is to have the DBMS perform the update directly. That is, the application program can instruct the DBMS to add one to the YEARS_OF_SERVICE component of each record. This results in no data being returned to the application program. However, the DBMS must now take the responsibility of insuring that all new values of YEARS_OF_SERVICE remain within the specified range. It is not acceptable for the DBMS to blindly change each value of YEARS_OF_SERVICE, only to have the application programs that retrieve the data at a later time discover that some values have become illegal.

3. Daplex

Data models and associated query languages have evolved significantly over the past two decades. The early hierarchical models were superseded by the network and relational models. The latter are in turn being superseded by so-called semantic data models. Our overall DBMS project is based on a semantically rich data model called Daplex which combines and extends the key features of earlier data models. For example, Daplex's modelling constructs are a strict superset of those found in the relational model. Daplex is designed to enhance the effectiveness and usability of database systems by capturing more of the meaning of an application environment than is possible with conventional data models. It describes a database in terms of the kinds of entities that exist in the application environment, the classifications and groupings of these entities, and the structural interconnections among them. The semantic knowledge captured in Daplex is not only meaningful to end users, but is also usable by the database system and database administrator for the purposes of query and physical schema optimization. For example, knowledge of the nature of relationships between types of entities (i.e., whether they are one-to-one, many-to-one, or many-to-many) can be used to control the appropriate clustering of entities of different types that are likely to be accessed together, both in a centralized and in a distributed environment.

The basic modelling constructs in Daplex are entities and functions. Entities correspond to conceptual objects. Entities are classified into entity types, based on the generic properties they possess. Functions represent properties of conceptual objects. Each function, when applied to an entity of appropriate type, yields a single property associated with that entity. Such a property is represented by either a single value or a set of values. These values can be simple, being drawn from Ada supported scalar types and character strings, or composite, consisting of references to entities stored in the database. We illustrate these constructs with an example.

Consider a university database modelling students, instructors, departments, and courses. Figure 1 is a graphical representation of the definition of

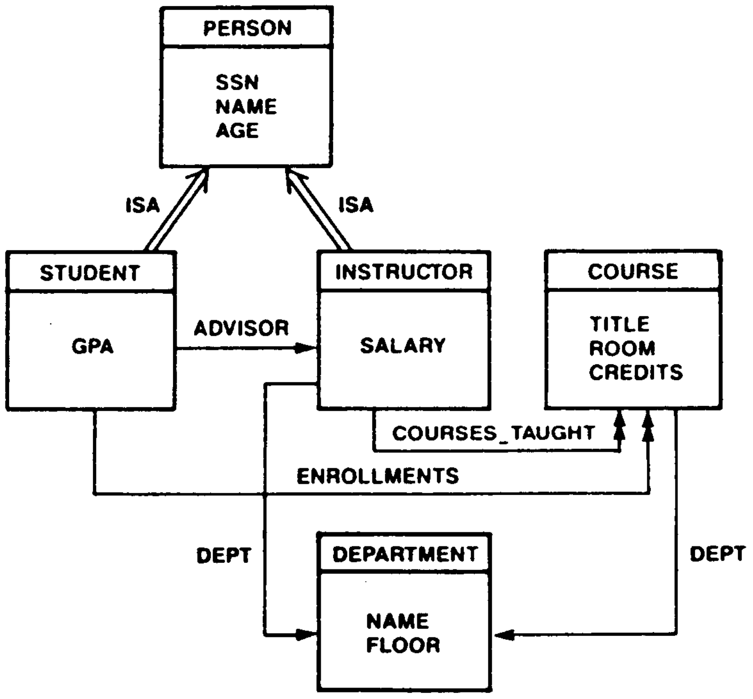


Figure 1. A Daplex Database

such a database. The rectangles depict entity types. The labels within the rectangles depict functions that range over Ada scalar and string types. The single-headed and double-headed arrows represent single-valued and set-valued functions that map argument entity types to result types. The double-edged arrows indicate isa (subtype) relationships.

One major difference between Daplex and the relational model is that referential integrity constraints [Date81], which are extremely fundamental in database applications but not easily specifiable in a relational environment, are directly captured. For example, when a student is inserted into the database, the database system will ensure that it is assigned a valid instructor, i.e., one that is existent in the database. Likewise, when an instructor is to be removed from the database, the database system will see to it that no dangling references result, i.e., there are no more students in the database who have the instructor in question as advisor.

Another important semantic notion captured in Daplex is that of a hierarchy of overlapping entity types. In relational systems, a real-world entity that plays several roles in an application environment is typically represented by tuples in a number of relations. In the university application environment, we might have an instructor entity named John Doe and a student entity also named John Doe. In this case, it might be desirable to impose the constraint that the age of John Doe as an instructor should agree with the age of John Doe as a student. One possible strategy in a relational system is to represent this information only once by having a relation person that stores the age information, and relying on joining operations to determine the age information for students and instructors. In Daplex, we can specify that student and instructor are subtypes of person whereby we can utilize Daplex's function inheritance semantics to simplify the formulation of queries and updates. Figure 2 shows a relational equivalent of the university database. Figures 3 and 4

```
PERSON (SSN, NAME, AGE)

STUDENT (SSN, ADV-SSN)

INSTRUCTOR (SSN, DEPT)

COURSE (ROOM, CREDITS)

ENROLLMENTS (SSN, TITLE)

COURSES_TAUGHT (SSN, TITLE)
```

Figure 2. A Relational Schema

shows a Daplex query and its equivalent in SQL [DATE84]. The intent of this query is to print the names of all students taking a class held at room "F320" and taught by an instructor in the "CS" department. Notice how explicit join terms have to be introduced in the SQL query, which tend to obscure readability. On the other hand, the absence of such constructs from the Daplex query allows the query to be read in a more or less English-like manner. A complete description of the Daplex data model and access language can be found in [SLRR84].

```
for each S in STUDENT where
    "F320" is in ROOM(ENROLLMENTS(S))
      and
    DEPT(ADVISOR(S)) = CS
loop
    PRINT(NAME(S));
end loop;
```

Figure 3. A Daplex Query

```
SELECT PERSON.NAME
 FROM   PERSON, STUDENT, ENROLLMENTS, COURSE, INSTRUCTOR
 WHERE PERSON.SSN = STUDENT.SSN
        AND PERSON.SSN = ENROLLMENTS.SSN
        AND ENROLLMENTS.TITLE = COURSE.TITLE
        AND COURSE.ROOM = "F320"
        AND STUDENT.ADV-SSN = INSTRUCTOR.SSN
        AND INSTRUCTOR.DEPT = CS
```

Figure 4. An Equivalent SQL Query

4. Adaplex

Database environments for popular programming languages, notably C, PL/1, COBOL, and Pascal, have resulted in extensions to the host programming language. At the outset, it was not clear whether Ada would also need to be extended to accommodate database applications. This is because Ada contains important new features not found in previous widely-used languages. In particular, Ada's package construct offers the potential for defining a database extension within the language itself.

There have actually been a number of proposals for coupling database management capabilities to Ada through the package construct [HTVN81, NOKI83, VINE83]. However, we feel that such approaches sacrifice usability and data integrity for not extending Ada [SCDF85]. Since our goal is to design the best Ada compatible language environment for developing database application programs, it is our desire to express as much of the database environment in Ada as possible, although not at the expense of database capabilities and ease of use.

Two major capabilities that must be provided by a database programming environment are schema definition (for describing the contents of the database) and transaction definition (for specifying operations on the stored data). In order to support database applications programming in Ada, it is necessary to couple the DBMS to an Ada programming support environment. One possible approach for achieving such a coupling is illustrated in Figure 5. Notice that both schema definition and transaction definition are separated from the Ada

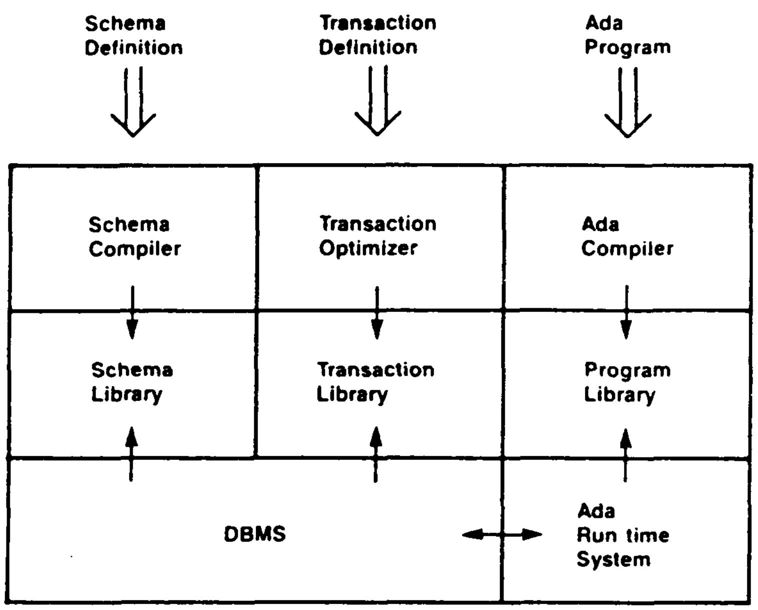


Figure 5. Coupling a DBMS with an Ada Programming Support Environment

application program.

This separation works for database schema definition since the output of the schema compiler can be logically thought of as an Ada package containing type definitions representing a database schema. The separation of transaction definition from application program is less natural because parameters must be passed from the application program to the DBMS and transaction results must be bound to application program variables.

In the course of our project, two approaches for handling transaction definition have been considered. The first approach is similiar to the one used for schema definition. A transaction definition is passed to the transaction optimizer which generates an Ada package that implements (i.e. calls the DBMS to execute) the transaction. The package is then loaded with the application program. This approach, however, leaves the applications programmer with a rather complicated interface. The programmer must learn a transaction definition language which is quite distinct from Ada. Besides, parameter passing between the application program and the package that implements the transaction is cumbersome. Since Ada is a strongly typed language, it might be necessary to use an intermediate representation like character strings for passing certain parameters. This has a number of drawbacks. First, the programmer must explicitly encode and decode these strings. Second, compile time type checking cannot be performed on the contents of these strings. In general, such a parameter passing mechanism can be quite inefficient.

These difficulties lead us to adopt a second approach which permits the application programmer to embed transaction definitions directly in an Ada program. The result is an integrated language, called Adaplex, which provides a tight coupling between Ada and our transaction definition language. No changes were made to existing Ada constructs. The new constructs that were added are treated in an Ada compatible manner. The coupling is achieved at the expression level. Applications programmers are free to use Ada

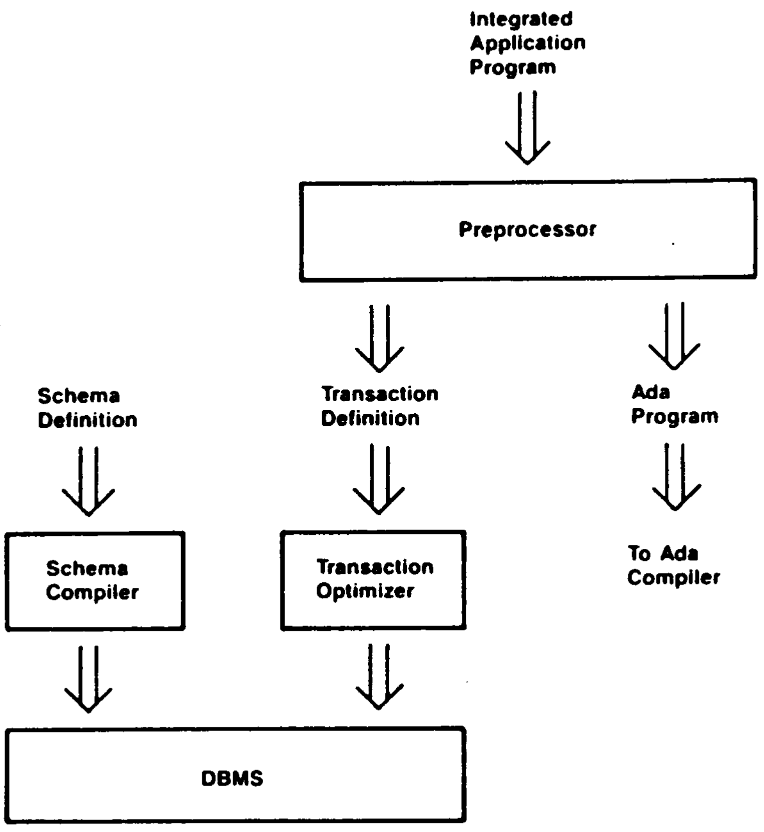


Figure 6. Configuration of Adaplex Programming Tools

expressions, control structures, and subprogram calls within a transaction definition. Because of Adaplex's uniform syntax and semantics, we expect it to be very easy to learn and use by trained Ada programmers.

For portability reasons, a preprocessor is used to decompose applications programs written in Adaplex into a transaction part and an Ada program part. The transaction part is forwarded to the transaction optimizer and the Ada part to the Ada compiler. The preprocessor is a very powerful tool. It provides the same integrity checking across the application program/DBMS interface that the Ada compiler provides for an Ada program.

The schema compiler, transaction optimizer, preprocessor, and DBMS form the minimum set of program development tools required for the database environment. Their combined configuration is shown in Figure 6. Any one of the Multibase, LDM, DDM systems can be substituted in place of the box labelled DBMS. Provided all these tools are written in Ada, database schemas, application programs, and databases may be ported between Ada installations.

Fundamentally, Adaplex adds two constructs to Ada, the *database declaration* and the *atomic statement*. These constructs provide for schema definition and transaction definition respectively. A database declaration specifies the data objects in a database, the types of those data objects, and their

```
database UNIVERSITY is

  type DEPT_NAME is (CS, EE, MA);
  type YEARS is new INTEGER range 0 .. 120;
  UNKNOWN_AGE : constant YEARS := 0;

  type COURSE is
    entity
      TITLE       : STRING (1 .. 6)
      ROOM        : STRING (1 .. 5);
      CREDITS     : INTEGER range 1 .. 4;
    end entity

  type PERSON is
    entity
      NAME        : STRING (1 .. 30);
      AGE         : YEARS := UNKNOWN_AGE;
      SSN         : INTEGER;
    end entity;

  subtype INSTRUCTOR is PERSON
    entity
      DEPT              : DEPT_NAME;
      COURSES_TAUGHT : set of COURSE;
    end entity;

  subtype STUDENT is PERSON
    entity
      DORM          : STRING (1 .. 10);
      ADVISOR       : INSTRUCTOR withnull;
      ENROLLMENTS : set of COURSE;
    end entity;

  overlap INSTRUCTOR with STUDENT;

  unique TITLE within COURSE;

end UNIVERSITY;
```

Figure 7. An Adaplex Database Declaration

consistency/integrity requirements. Database declarations are processed by the schema compiler. Figure 7 shows the database declaration for the university database that was depicted graphically in Figure 1. In addition to the type and subtype declarations, several constraint statements have been specified.

```
overlap INSTRUCTOR with STUDENT;
```

indicates that it is legal for a PERSON entity to be both a STUDENT and INSTRUCTOR simultaneously.

```
unique TITLE within COURSE;
```

indicates that all COURSE entities must have unique TITLEs.

```
with UNIVERSITY; use UNIVERSITY;
  .
  .
  .
ADD_COURSE:
  declare
    NEW_COURSE : COURSE;
  atomic
    NEW_COURSE := new COURSE (TITLE   => "CS-101",
                              ROOM    => GET_ROOM(CS),
                              CREDITS => 3);
    include NEW_COURSE into
      COURSES_TAUGHT
        (I in INSTRUCTOR where NAME (I) = "Adam Jones");
  exception
    when UNIQUENESS_CONSTRAINT =>
      PUT_LINE("Duplicate course name");
  end atomic;
```

Figure 8. An Adaplex Database Transaction

A database is similar to a package since it is a related collection of data and type declarations. However, a database differs from a package in three principal ways. First, there are explicit protocols within Adaplex for several independent main programs to share the use of a database. Second, a strong discipline is imposed on the specifications allowed in a database declaration. Third, database declarations are developed interactively via the schema compiler, and they are stored for future reference in the schema library.

An atomic statement specifies a compound operation which must be indivisibly executed with respect to a database. The preprocessor extracts transactions from atomic statements for processing by the transaction optimizer. Figure 8 shows an Ada code fragment containing an atomic statement. This transaction creates a new COURSE entity and indicates that the course will be taught by the instructor named Adam Jones. Notice that the database type declarations are made visible by the with and use statements. The expression level integration of Daplex and Ada is illustrated by calling an Ada subprogram, GET_ROOM, to generate a value to assign to the ROOM function. Since COURSEs are constrained to have unique TITLEs, it is possible that the create statement may fail. An exception handler is included to cleanly handle this error.

An atomic statement is similar to a block in the sense that it is a compound statement that has associated declarations and exception handlers. However, an atomic statement differs from a block in three ways. First, atomic statements are executed indivisibly with respect to databases. Second, strong disciplines are imposed on the contents, nesting, parallel execution, and exception handling of atomic statements. Third, atomic statements are transformed by the preprocessor to extract database transactions.

A complete description of the Adaplex language can be found in [SFL83]. A detailed discussion on our rationale for developing Adaplex can be found in [SFL83, SCDF85].

5. LDM

LDM is a general purpose system for defining, storing, retrieving, updating, sharing, and protecting formatted information. While its users may be geographically distributed, LDM and its data must be centrally located. LDM is designed to provide all the functions typically found in a modern database system, including:

- logical and physical database definition,
- logical and physical database reorganization,
- a fully integrated data dictionary facility,
- an authorization mechanism for controlling database access,
- optimized selection of access paths for transactions,
- interference-free concurrent access by multiple users/transactions,
- automatic recovery from transaction failures, software crashes, and media failures,
- a dumping utility for taking a consistent snapshot of the entire database,
- a reload utility for restoring a database to a previously saved state.

LDM's main design objectives are transportability and high performance. Transportability is achieved by the use of Ada as the implementation language and by using a modular system architecture which is greatly facilitated by Ada's packaging construct and separate compilation mechanism. A description of LDM's component architecture can be found in [CFLR81]. High performance, on the other hand, requires the introduction of a number of technical innovations in the areas of physical data structuring, query optimization, concurrency control, and recovery management as identified below.

LDM is designed to provide complete physical data independence. It supports flexible physical structuring options so that a database administrator can tailor the physical representation of a database according to application requirements [CDFL82]. LDM employs special data structures for the efficient maintenance of referential integrity and other contraints associated with type overlaps in a generalization hierarchy. It also provides a wide range of options for the clustering of entities that belong to a generalization hierarchy. LDM supports dynamic data structures (namely, linear hashing [LARS80] and B-trees [COME79]) to eliminate the need for periodic reorganization. In order to support the efficient traversal of interentity references, LDM implements a pointer validation scheme that minimizes the updating costs associated with the use of dynamic data structures.

The design of LDM is geared towards the processing of repetitive transactions in a database applications programming environment. Transactions are compiled, thereby permitting the costs for parsing, authorization checking, and access path optimization to be amortized over multiple execution. LDM is also designed to optimize a much larger class of queries than relational systems. In particular, we have developed efficient strategies for processing queries with outerjoins and nested quantifiers [RCDF82, DAYA83A]. At the same time, the amount of effort that LDM will expend to optimize a transaction template can be controlled by a user (in the form of a pragma). Thus, a user can ensure that the effort for optimizing a given transaction template is commensurate with the savings that can be expected to accrue over repeated execution.

LDM implements an integrated concurrency control and recovery mechanism which has the advantage of improving concurrency while simplifying transaction and system recovery. Specifically, LDM implements a multiversion mechanism that allows each read-only transaction to see a consistent snapshot of the database without having to synchronize with update transactions [CFLN82]. The essence of this mechanism is that update transactions create new versions of data objects without overwriting their previous versions. An efficient scheme is used to determine the appropriate version of different data objects each read-only transaction should see, and to identify those old versions that can be garbage collected. Since database dumps can be considered as read-only transactions that access the entire database, they can also be taken non-intrusively (i.e., without requiring the quiescence of concurrent updates).

In addition to being a stand-alone centralized database system, LDM also functions as an integral part of DDM and Multibase.

6. DDM

DDM is a homogeneous distributed database system built on top of a collection of LDMs running at different sites connected by a computer network. From the end-users' point of view, DDM performs precisely the same operations supported by LDM. This is because all complexities introduced by fragmentation, distribution, and replication of a database are hidden from end-users. Users access a distributed and replicated database in DDM just as they would access a centralized database in LDM. In a distributed environment, a copy of LDM and a copy of DDM are installed on each of several computers in a computer network where data is distributed / replicated. Each LDM is responsible for managing all locally stored data at its resident site. Each DDM cooperates with all other DDMs in the network in order to hide the distribution and replication of data from end users and applications. As a truly distributed system, DDM delivers the benefits of improved processing capacity, communications efficiency, survivability, and modular upward scaling. DDM provides the following important facilities.

- An integrated global schema that encompasses data stored at all sites. DDM maintains a global directory in order to keep track of the distribution and replication of data. It automatically maps transactions on the global schema into subtransactions on data stored at individual LDMs.
- Complete physical data independence. The database administrator is free to tune parameters involving the physical distribution, replication, and representation of the stored data, without affecting the external view of the database.
- Mutual consistency of replicated data. Users deal with logical data only. Propagation of updates to redundant copies of updated data is managed by the system.
- Atomicity of distributed transactions. DDM guarantees than no partial effects of one transaction will be seen by another. If a transaction is unable to complete, all of its effects on the database are automatically undone.

- Continued operation in spite of site failures. Users can continue to perform retrieval and update operations, even though some copies may be temporarily inaccessible. These latter copies are brought up to date by the system before being used for processing subsequent transactions.
- Dynamic integration of new sites. No quiescence of on-going activities is needed for reconfiguration of the system.

As in LDM, our main design objectives for DDM are transportability and performance. Again, we have introduced a number of technical innovations in the areas of data allocation, query optimization, concurrency control, and recovery management in order to obtain good performance. These are summarized below.

DDM supports flexible database fragmentation and allocation that can be used to improve locality of reference and efficiency of query processing [CDFR83]. Each database managed by DDM is optionally divided into a number of groups of data fragments, based on the likelihood of their being used together. Each group of data fragments constitutes a unit for allocation and may optionally be replicated at as many sites as desired. For a replicated fragment group, two kinds of copies are distinguished. Online copies are used for processing transactions. Offline copies serve as warm standbys that can quickly (and automatically) be upgraded to online status in order to retain a desired degree of resiliency as sites storing online copies fail. When specifying the replication parameters for a fragment group, a database administrator indicates the number of desired online copies and those sites whose copies are to be kept online preferrably. DDM will then strive to keep those copies at the preferred sites online, but dynamically bringing copies stored at other sites online to maintain the desired level of resiliency when necessary.

Unlike previous systems, DDM is designed to take into consideration database fragmentation and replication in its selection of strategies for processing transactions [CDFG83]. Whereas most previous studies on distributed query optimization assume the distribution of joins over unions, DDM will consider the options of using left distribution, right distribution, or no distribution at all when processing queries that involve such operations. DDM treats each fragment group as an integral data unit during the optimization process. Both compile time and run time optimization are performed. Compile time optimization seeks to identify a good order for processing the high level data manipulation operations on fragment groups without binding operations and copies to sites. This is because the choice of which copy of a fragment group to use for processing a transaction cannot be made until the availability of sites at run time is known. By dividing the optimization into two stages, DDM maximizes the amount of preanalysis done at compile time while ensuring the validity and optimality of the generated access plans.

DDM's concurrency control mechanisms are extensions of those used in LDM. Again, a multi-version mechanism is used to eliminate conflicts between read-only and update transactions [CG85]. In addition to improving parallelism, this mechanism greatly facilitates the taking of global checkpoints. Such a checkpoint may be necessary if one wants to reset a distributed database to a previous globally consistent state after the log data in one or more sites is damaged. With respect to replica control, DDM provides a balance between synchronization overhead and failure resiliency. Essentially, updates are propagated to online copies synchronously. Offline copies are only updated in a background batched fashion.

Because DDM is designed for distributed command and control applications, survivability is a very important issue. A special transaction commit algorithm is used to ensure that distributed transactions are terminated in a timely fashion, even in the presence of site failures, so that resources at the remaining operational sites can be fully utilized (without being tied down by incomplete transactions). DDM is designed to recover automatically from total failures wherein all of the sites coordinating a transaction or all of the sites storing replicated copies of a fragment group fail simultaneously. Previous systems have treated such failures as catastrophes and required human intervention for recovery. In order to speed up the availability of data at a recovering site, DDM employs an incremental site recovery strategy. Essentially, the fragment groups stored at the recovering site are prioritized and brought up to date one at a time (with the assistance of other replication sites). As soon as a fragment group is brought online, it can be used for processing new transactions without having to wait for the recovery of other fragment groups.

7. Multibase

Multibase is designed to provide a logically integrated, retrieval-only, user interface to a physically nonintegrated environment containing pre-existing databases. These databases may reside on different types of database management systems, at different physical locations, and on different types of hardware.

Before local databases can be accessed through Multibase, the local host systems must be connected to a communications network. This network can be local or geographically distributed. After Multibase has been connected to the same communications network, a global user can access data in the local databases through Multibase using a single query language. Each local site maintains autonomy for local database updates. Local applications can continue to operate using the existing local interfaces, as before.

Multibase presents the end user or application program with the illusion of a single, integrated, non-distributed database. Specifically, Multibase assumes the following responsibilities:

- providing a global and consistent picture of the available data,
- knowing the locations for the database items,
- transforming a query expressed in the global query language into a set of subqueries expressed in the different languages supported by the target systems,
- formulating an efficient plan for executing a sequence of subqueries and data movement steps,
- implementing an efficient plan for accessing the data at a single target site,
- moving the results of the subqueries among the sites,
- resolving incompatibilities between the databases (such as difference in naming conventions and data types),

- resolving inconsistencies in copies of the same information that are stored in different databases, and
- combining the retrieved data to correctly answer the original request.

Multibase has three key design objectives: generality, compatibility, and extensibility. To satisfy the first objective, Multibase has been designed to be a general tool, capable of providing integrated access to various database systems used for different applications. Multibase has not been engineered to be an interface for a specific application area. The second requirement of Multibase is that it co-exists and be compatible with existing database systems and applications. No changes or modifications to local databases, DBMS's, or application programs are necessary to interface Multibase with systems already in operation. The local sites retain full autonomy for maintaining the databases. All local access and application programs can continue to operate without change under Multibase. The third design objective is that it must be relatively easy to couple a new local system into an existing Multibase configuration.

All these objectives are achieved by designing a modular architecture for Multibase and by making the system largely "description driven" [LR82]. Multibase's modular architecture isolates those parts of the system that deal with specific aspects of a local system. Because of this, a Multibase configuration can be expanded to include a new DBMS in a short period of time and with little impact on the existing Multibase software. Descriptions are used throughout Multibase to tailor general modules for specific applications, users, and databases. These descriptions are written by the database administrator(s) who is responsible for tailoring a Multibase configuration.

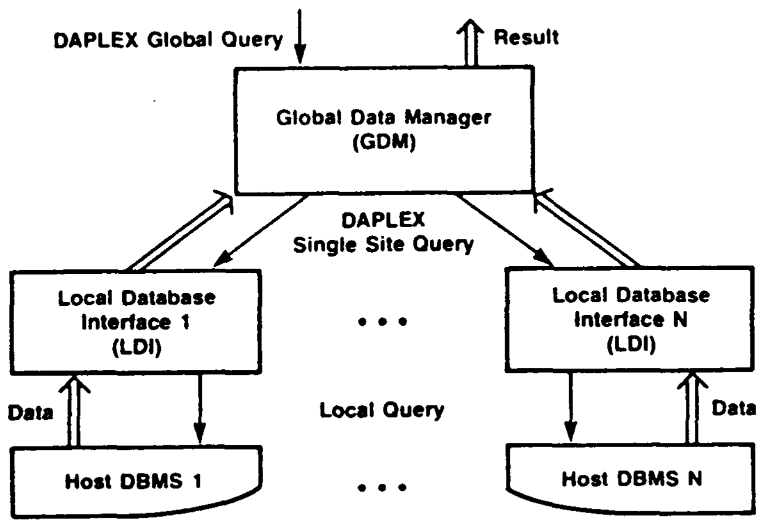


Figure 9. Multibase Component Architecture

The component architecture of Multibase is illustrated in Figure 9. There are two types of modules: a global data manager (GDM) and a local database interface (LDI). All global aspects of a query are handled by the GDM. All specific aspects of a local system are handled by an LDI. There is one LDI for each local host DBMS accessed by Multibase. The GDM makes use of LDM as an internal DBMS to manage its workspace. The LDM is used to store the results of the Daplex single-site queries which are processed by the LDI's and to perform all the required steps of the final query for combining and formatting the data.

It should be mentioned that Multibase does not provide the capability to update data in the local databases or to synchronize read operations across several sites. This is because implementing global concurrency control mechanisms for read or update operations would have necessitated the global process to request and control specific resources offered by the local systems (i.e., locking local database items) as required to ensure consistency across the databases. However, most systems do not make available to an external process the services necessary to implement global concurrency control. Since Multibase is designed to operate without requiring modifications to existing systems, the tools necessary to ensure consistency across databases are not globally available. Thus, autonomy of database update is maintained locally, and Multibase provides the global user with the same level of data consistency that the local host DBMSs provide to each local database user.

In addition to the highly modular and description driven architecture, the design of Multibase has required research in the areas of schema integration, global query optimization, and local query optimization. Our results in each of these areas have been reported in [KG81, DAYA84a], [DAYA83b, GY84, DAYA84b], and [DG82] respectively.

8. Status

Designs of the Daplex and Adaplex languages are complete. Prototype versions of Multibase and LDM which support most of the described capabilities have been implemented. Implementation of DDM is well underway. To date, the systems contain approximately 500,000 lines of Ada source code. Most of the implementation was done in an Ada-subset using an Ada-to-Pascal translator [SOFT81]. The systems were then converted to full Ada using the DEC VAX Ada compiler [DEC85]. Development is continuing using both VAX Ada and Rational's Ada Development Environment [RAT85]. The initial target environment for all three systems is VAX VMS. The current systems support an interactive version of Adaplex (i.e., Daplex).

9. References

[CDFG83b]
A. Chan, U. Dayal, S. Fox, N. Goodman, D. Ries, D. Skeen. "Overview of an Ada Compatible Distributed Database Manager." *ACM SIGMOD Conference Proceedings*, 1983.

[CDFL82]
A. Chan, S. Fox, S. Danberg, W. Lin, A. Nori, D. Ries. "Storage and Access Structures to Support a Semantic Data Model." *VLDB Conference Proceedings*, 1982.

[CDFR83]
A. Chan, U. Dayal, S. Fox, D. Ries. "Supporting a Semantic Data Model in a Distributed Database System." *VLDB Conference Proceedings*, 1983.

[CFLN82]
A. Chan, S. Fox, W. Lin, A. Nori, D. Ries. "The Implementation of an Integrated Concurrency Control and Recovery Scheme." *ACM SIGMOD Conference Proceeding*, 1982.

[CFLR81]
A. Chan, S. Fox, W. Lin, D. Ries. "The Design of an Ada Compatible Local Database Manager." Technical Report CCA-81-09, Computer Corporation of America.

[CG85]
A. Chan, R. Gray. "Implementing Distributed Read-only Transactions." To appear in *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, February 1985.

[DATE81]
C. Date. "Referential Integrity." *VLDB Conference Proceedings*, 1981.

[DATE84]
C. Date. *A Guide to DB2*, Addison Wesley, 1984

[DAYA83a]
U. Dayal. "Processing Queries with Quantifiers: A Horticultural Approach." *ACM PODS Conference Proceedings*, 1983.

[DAYA83b]
U. Dayal. "Processing Queries over Generalization Hierarchies in a Multidatabase System." *VLDB Conference Proceedings*, 1983.

[DAYA84a]
U. Dayal, H. Hwang. "View Definition and Generalization for Database Integration in Multibase: A System for Heterogeneous Distributed Databases." *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, November 1984.

[DAYA84b]
U. Dayal. "Query Processing in a Multidatabase System." in *Query Processing in Database Systems*, (W. Kim, D. Batory, D. Reiner, editors), Springer Verlag, 1984.

[DEC85]
Digital Equipment Corporation. "Developing Ada Programs on VAX/VMS." 1985.

[DG82]
U. Dayal, N. Goodman. "Query Optimization for CODASYL Database Systems." *ACM SIGMOD Conference Proceedings*, 1982.

[GSCD83]
N. Goodmam, D. Skeen, A. Chan, U. Dayal, S. Fox, D. Ries. "A Recovery Algorithm for a Distributed Database Management System." *ACM PODS Conference Proceedings*, 1983.

[GY84]
D. Goldhirsh, L. Yedwab. "Processing Read-Only Queries over Views with Generalization." *VLDB Conference Proceedings*, 1984.

[HD84]
H. Hwang, U. Dayal. "Using Semiouterjoins to Process Queries in a Multidabase System." *ACM PODS Conference Proceedings*, 1984.

[HTVN81]
J. Holland, K. Tai, M. Van Name. "An Ada Relational Database Interface Using Abstract Data Types." TR 81-07, North Carolina State University, 1981.

[KG81]
R. Katz, N. Goodman. "View Processing in Multibase — A Heterogeneous Database System." in *Entity-Relationship Approach to Information Modelling and Analysis*, (P. Chen, editor), ER Institute, Saugus, CA, 1981.

[LR82]
T. Landers, R. Rosenberg. "An Overview of Multibase." in *Distributed Databases*, (H. Schneider, editor), North Holland Publishing Company, 1982.

[NOKI83]
Nokia Data Terminal Systems. "MPS 10 Database Management System Functional Description." Version 1.0, June 1983.

[RAT85]
Rational, Inc. "Rational Environment Reference Summary." 1985.

[RCDF82]
D. Ries, A. Chan, U. Dayal, S. Fox, W. Lin, L. Yedwab. "Decompilation and Optimization of Adaplex: A Procedural Database Language." Technical Report, Computer Corporation of America, 1982.

[SCDF85]
J. Smith, A. Chan, S. Danberg, S. Fox, A. Nori. "A Tool Kit for Database Programming in Ada." To appear in *International Ada Conference Proceedings*, 1985.

[SFL83]
J. Smith, S. Fox, T. Landers. "Adaplex: Rationale and Reference Manual." Technical Report, Computer Corporation of America, 1983.

[SLRR84]
S. Fox, T. Landers, D. Ries, R. Rosenberg. "Daplex User's Manual." Technical Report CCA-84-01, Computer Corporation of America, March 1984.

[SOFT81]
SofTech, Inc. "Interim Ada-to-Pascal Translation Tool Language Reference Manual." TP 124, September 1981.

[SWK76]
M. Stonebraker, E. Wong, P. Kreps. "The Design and Implementation of INGRES." *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[VINE83]
D. Vines, Jr. "An Interface to an Existing DBMS from Ada (IDA)." GTE Network Systems, 1983.

N89-16372

A Study of MAPSE Extensions

David Auty, SofTech, Inc.
Robert Charette, SofTech, Inc.
Charles McKay, UHCL High Technology Laboratory

1. Overview

This project was initiated to study the technical issues of extending the MAPSE to support the life cycle of large, complex distributed systems such as the Space Station Program (SSP). The work has been divided into two phases. Phase one, covered by this report, identifies a list of advanced technical tools needed to extend the MAPSE to meet the needs believed to be inherent in the Software Support Environment (SSE). Of secondary importance was the identification of a list of advanced management tools.

Phase two, which is on-going at this time, is to study and document the major technical issues in adding these tools to the MAPSE as an integrated extension evolving into an appropriate SSE. The intent is to provide a framework for understanding and evaluating the subsequent development and/or procurement of such tools.

This paper has been extracted from the full interim report on the phase one efforts. It incorporates just the description of SSE requirements, and a list of the tools identified. Other topics addressed in the interim report include an outline of the principle requirements for a MAPSE, a description of the life cycle model and a description of the tools in the context of the life cycle model.

For the purpose of this paper, the basis life cycle model is an adaptation of the symbolic representation of McDermid and Ripken (1984) to the model described in DoD Standards 2167 and 2168. The model partitions the process of software development into the following phases:

p1: System Requirements Analysis,
p2: Software Requirements Analysis,
p3: Preliminary Design,
p4: Detailed Design,
p5: Coding and Unit Test
p6: Computer Software Component Integration

The outputs from each phase are the formal review documents used for verification and validation, which also form the inputs to the succeeding phases. All documents and development information are maintained in an integrated life cycle project object base which serves to centralize and control the development process. All activities and tools work with this project object base to maintain the parallel processes of configuration and quality control.

2.0 A Brief Description of Support Environment Requirements in the Context of the Life Cycle Model

2.1 System Requirements Analysis

2.1.1 Characteristics, Principles and Methods

Several activities should be pursued during requirements interpretation, feasibility studies, and analysis.

Semantic Information Capture - Supporting interpretation, the capture of requirements in the form of a semantic model involves identifying key terms, categorizing the terms, defining the terms, and identifying the relations between the terms. The capture of semantic information creates a recording of the semantic model of the requirements, which becomes part of the baseline. Assuming the semantic information is machine-encoded, it might be expressed in a formal language such as Problem Statement Language (PSL) or in a combination of formal graphics and text expression such as Software Requirements Engineering Methodology (SREM).

Semantics Analysis - Once the requirements are expressed in the context of a semantic model, the model relations can be used for a systematic analysis of the completeness and consistency of the requirements. This is achieved by asking questions which are answered with the aid of the relations, such as "Are there any other processes which should be related to Process A by the 'predecessor of' relation?"

Traceability may be established through reference relations between requirements and specification, design and code, etc. The relational analysis can be used to assess the impact of requirements changes on the baselined products.

The semantic analysis activity aids development by identifying areas of requirements incompleteness or inconsistency.

Feasibility and Risk Analysis - Evaluating the feasibility of requirements is a significant part of requirements analysis. Feasibility should be viewed from the perspectives of design, performance and cost.

Design feasibility involves finding at least one design that satisfies the requirements. Any approach from trial design to prototyping is appropriate. Performance feasibility is a special case of design feasibility analysis. Once a trial design is established, modeling is an effective technique for analyzing performance. Cost feasibility involves estimating costs based on the trial design. Cost analysis must consider the three key elements: the development phase, the operations phase, and the phase for continuing adaptation.

2.1.2 Requirements on the Support Environment

The requirements on the support environment data base, derived from requirements analysis, are:

Baselined Products - The semantic information is the only data associated with requirements analysis that should be baselined. It should be under configuration control and subject to change only as requirements changes are approved. Baselined data should not only include the "shalls" of each phase (which must be dichotomously demonstrated at acceptance test time) but also the "shoulds" which have life cycle implications that cannot be dichotomously demonstrated at acceptance test time and which may require the design of special metrics and instrumentation to support their analysis at subsequent points in the life cycle.

Non-Baselined Data - Any information associated with modeling, simulation, prototyping, or semantic analysis should be saved temporarily. It should be used later in requirements analysis iteration or other activities.

Measurement Data - Several measurements of the requirements analysis activity and its outputs should be captured:

- Size of the data base for semantic information,
- Complexity of the requirements as measured by the relationships in the semantic information and
- Number of inconsistencies or omissions found.

2.2 Software Requirements Specification

2.2.1 Characteristics, Principles and Methods

Formal Recording - The specification information must be recorded in some suitable form. As a minimum, the specification should describe interactions, modes and functions. It should have the characteristics of being minimal, understandable, accurate and precise, and easily modified. The specification should use a formal notation to facilitate formal correctness analysis and automated analysis of the specification easier.

Completeness Analysis - This is done by trying out a design of the system. Completeness analysis generates questions which can help identify information absent from the requirements. In many cases, this activity is done during requirements analysis.

Correctness Demonstration - The specification must be shown as consistent with the requirements. Since the requirements may not be stated in a formal manner, a rigorous proof of their consistency may not be possible. The correctness demonstration is then produced through a subjective, informal analysis based on the semantics information from requirements analysis.

Consistency Analysis - Any method for performing this analysis depends on the form of the specification information. If a formal specification language is used, certain kinds of problems may be detected by analyzing this notation. In other cases, the consistency of the specification information must be judged on a less precise basis. A good example of the state-of-the-art in specification methods is that advocated by the Naval Research Laboratory, and used to develop the specification of the A-7 flight program. The specification document includes formal, tabular notation which lends itself to completeness and consistency analyses.

2.2.2 Requirements on the Support Environment

The requirements on the support environment data base, derived from the specification activity are:

Baselined Products - The specification information is baselined. Any modeling information produced should be baselined if it is crucial to the life cycle support of the software.

Non-Baselined Data - This material includes partial specifications under development, alternate specification, and diagnostic information produced by specification analysis tools.

Measurements - Examples of useful measurements data to be captured are: effort and resource data concerning the development of the specification, size data, number of errors and changes made, and subjective measures of the quality and completeness of the specification.

2.3 Preliminary and Detailed Design

2.3.1 Characteristics, Principles and Methods

A design is the translation of the "shalls" from requirements analysis into Ada package specifications. Functional requirements should be transformed into functional Ada specifications that can be checked by an Ada Compiler. Non-functional requirements (i.e., constraints) should be transformed into a discipline of Ada comments that can be checked by other APSE tools.

Three areas of design support are identified: formal recording of system design, formal recording of data and program design, and creative aids.

Formal Recording of System Design - There are several methods involved in recording the system design.

Information-Hiding - This method involves isolating information within modules. The module limits are defined by the information (design decisions, data definitions, etc.) to be isolated. Design is based on the expected changes to the information, thus localizing the effect of future changes.

Module Specification - Focusing on module specifications yields a description which allows others to determine the intent of a complete module by reading the module specification.

Use Hierarchy - Focusing on the use hierarchy yields a description which explains which programs depend on the correct implementation of a given module to produce correct results.

Formal Recording of Data and Program Design - The techniques and methods for the formal recording of data design and program design are:

Program Design Language (PDL) - The writing of data and program design in a PDL is a useful technique for formally recording the program design. It is sufficiently low-level to support direct coding, and is flexible enough to leave some questions unanswered while the designer proceeds with the design. (i.e., Ada Source code with Ada "stubs".)

Stepwise Refinement - This method goes hand in hand with PDL. With stepwise refinement, specifications for the lower level code become part of the documentation of the procedure. This makes the intent of the code much clearer.

Abstraction of Data Types - With abstraction, the designer can develop details where they are needed. This permits information-hiding as well as a more independent implementation of the system.

Creative Aids - Many creative techniques exist for design. A designer chooses techniques based on their individual approach to creativity. Some prefer graphic techniques while others do not. The choice of creative techniques should be left to the individual, whereas the techniques for formal recording must be standard. Described below are some representative creative aids:

Data and Control Flow Analysis - Module decomposition and function allocation are based upon the data and control flows required by the system. An example is Structured Design.

Data Structure Transformation - Transformation is a design technique in which the structure of the input and output data determines the structure of the program.

Graphic Decomposition Techniques - Graphs showing hierarchic relations depict the decomposition at many levels. An example is Structured Analysis and Design Technique (SADT).

Graphic Control Descriptions - Other ways of showing the control flows in the program are Petri Nets and Warnier-Orr diagrams.

2.3.2 Requirements on the Support Environment

As with the other activities of development, the data base must contain information on the design.

Baselined Products - Throughout the life of the system, the most recently approved form of the design must be stored in the data base. The system design is entered before the design of various subsystems or modules.

Non-Baselined Data - This includes preliminary designs as well as graphic displays used during the creative process. Graphic displays include tree structures, block diagrams, and other material created by design tools. The data base must provide for maintaining the temporary designs developed before one is actually chosen and baselined.

Measurements - These should include module interconnection measurements, such as data bindings. These should also include lower design measurements, such as cyclomatic complexity, and operators and operands. Many of these measurements are normally taken on the completed code, but with good, low-level PDL, they can be taken (or approximated) during design.

Archival Data - Archived data should capture the motivation behind the choice of design. The archived data should also include past designs evolved from use or rejected during development along with the reasons for the rejection.

2.4 Coding and Unit Test, Computer Software Component Integration

2.4.1 Characteristics, Principles and Methods

This section will focus on the unique requirements of developing distributed systems.

Designs which map program entities across distributed processing resources should be specified in two complementary parts. First, the functional requirements should be demonstrated to be met by the program design by executing the program in the host environment. (I.e., compile and execute the Ada source code on the host system without regard to properties of distribution.) Second, the non-functional requirements (i.e., constraints) such as the location each program entity is to be assigned, timing constraints, sizing constraints, etc. should be mapped to a simulator for analysis of the implications of imposing these restrictions upon the design which was proven in the first step. Tuning of assignments, code, algorithms and structures can take place in the host environment until the simulator provides a degree of confidence. Load modules can then be built and moved to the target environment or to a target test bed for further study. The implementation should produce an effective, understandable transformation of the design. The automatic generation of appropriate comments in the source code can ease the more complex process of maintenance in a distributed environment.

The following are some key aspects of implementation:

Standard Interface Set to a Catalog of Runtime Support Environment Features and Options. - This interface set establishes a virtual Ada machine. The compilation system produces target code that uses the services provided by the standard interface set. The requested service determine which modules of the runtime support library are to be exported to the target environment.

Target Network Topology Specification - This allows the designer to specify the symbolic names for remote area networks, local area networks, and individual processing nodes. The design also identifies the communications support available to link the various entities of the network.

Target Node Resources Specification - This allows the designer to specify the hardware resources for each node identified with the network topology specifier. The system will retain this information in the project object base along with the collection of software resources that will be assigned to this node later in the design. The designer declares the instruction set architectures available, the memory banks and their attributes, the buses and their attributes, and the communications links that are available.

Partitioning and Allocation Specification - After the Ada source code has been transformed into a DIANA representation and executed to demonstrate that it meets the functional requirements of the program, a discipline of comments and key words such as "location" can be used to map each program entity to a symbolic location. This symbolic location corresponds to those node and network identifications previously entered with the topology specification and the node resources specification. These non-functional requirements are added as attributes to the DIANA representations.

Distributed Workload Simulation - After the symbolic location assignments and other constraints have been added to the attributes of the DIANA representation, the workload simulator examines the project object base to determine characteristics of the already existing workload (if any) and to select empirical estimates of communications delays, processing throughput, and other relevant estimators. A simulation is then provided for analysis. If the analysis indicates the design approach is not feasible, new approaches to distribution can be provided by returning to the partitioning and allocation specification.

Distributed Program Building - When the workload simulation indicates a feasible design, the process of building new load modules includes examining the symbolic location assignments added to the DIANA tree and looking these up in the project object base to determine what type of instruction set architecture the particular entity's object code is to be generated for. If the code is to be added to the workload of an existing system, it is also necessary to identify if additional modules or new versions of the run time library need to be added or if additional hardware is likely to be needed to accommodate the increase in workload. The end result of the program building activity is to prepare a load module consisting of applications code and the necessary support from the run time library for each of the processors affected by the distribution of the program entities.

Run Time Support Environment Monitoring - If life and property are to depend upon the program meeting both its functional and its non-functional requirements, it may be desirable to prepare the program for execution in a target testbed. to be effective, the testbed should be fully instrumented and interact with the host environment. This requires the support of a run time monitor for each processor in the target testbed to interact with the instrumentation and host environment to provide meaningful information.

2.4.2 Requirements on the Support Environment

The most important requirements and opportunities for the support envrionment life cycle project object base become evident from this phase. The results are summarized below:

Baselined Products - The functional requirements are similar to those described in the preceding sections. However, opportunities arise due to the requirements for the DIANA representation in the implementation phase. An estimated ten to twenty times the processing time is required to convert Ada source code to DIANA representation as compared to converting the DIANA representation to object code for the target environment. Furthermore source code and object code can both be reconstructed from the DIANA representation. Since the Stoneman requirements for the MAPSE provides a unique identification for each object produced (which includes history attributes identifying the time, date, tools, etc. used to manipulate the object), an enormous amount of on-line storage space can be conserved in the project object base if the DIANA representation is maintained as the baseline.

The other important implication for baseline control as a result of this phase is the identification and maintenance of the network topology and the network node resources described in the preceding section

Non-Baselined Data - The temporary storage required for this category is similar to the functional requirements listed in the other sections of this report. However, the savings and storage space made possible by the utilization of DIANA representation described above may be significant even for temporary storage requirements.

Measurement Data - A number of metrics regarding the utilization of these tools is desirable. Knowing who is using the tools for what projects, and knowing the frequency of reference can provide valuable management insights.

2.5 Verification and Validation

2.5.1 Characteristics, Principles and Methods

The methods linked with correctness analysis are either static analysis or dynamic analysis. Static analysis includes, in order of increasing rigor, reviews, inspections, and proofs of correctness. Dynamic analysis includes all testing techniques.

Reviews - Reviews determine the internal completeness and consistency of system requirements and software specification, design and test information. They also assess its consistency with its predecessor information. Reviews involve a broad range of people, including developers, managers, users, and outside experts or specialists. A review must have specific objectives and questions to be addressed. The review findings generate rework tasks for the development group.

Inspections - Inspections evaluate the correctness of component level specification, design, code, test plans, and test results. They are more formal and rigorous than reviews. An inspection involves a small group of people of a specific make-up, and follows a well-defined procedure.

Proofs of Correctness - All development products should be verified with an informal proof of correctness. Certain critical kernels of code or special applications may require a formal proof of correctness.

Testing - Dynamic execution of the system or system component with known inputs in a known environment is a "test". If the test result is consistent with the expected result, the component is deemed correct in the limited context of the test. The following baselined documents are created relative to testing:

- Test Plan - Defines the scope, approach, and resource needed for testing.

- Test Procedures - Provides a detailed description of the steps and test data associated with each test case.

- Test Results - Documents the results of each test run. Unsuccessful runstrigger trouble reports which must be addressed by the development group.

The relationships between system functions and component or system test cases should be clearly established. Then, when changes are made to parts of a system, a subset of test cases can be identified which will test the system sufficiently. This process is called regression testing. Effective regression testing is a good way to reduce software development costs.

2.5.2 Requirements on the Support Environment

The requirements on the Support Environment data base, derived from the correctness analysis, are summarized below:

Baselined Products - Test plans, test procedures and test results (of correctly executed tests) are all baselined. They are controlled by configuration management. The results of inspections and proofs might also be baselined.

Non-Baselined Data - The non-baselined data includes work-in-progress, static analysis data, trouble reports, and debug data. Temporary storage of this type of information is required.

Measurement Data - A number of measurements associated with correctness analysis should be captured. These include: number of modifications to a unit, number of errors found per unit, number of test runs, number of errors by error category, and test coverage.

2.6 Project Management Support

2.6.1 Characteristics, Principles and Methods

Estimation - Most resource estimation techniques use the measurements from prior projects to estimate resources. Support of estimation methods requires a data base of comprehensive measurements including such software system parameters as size of source code, source language, development resources expended, and complexity measures.

Precedence Networks - This planning method is used to analyze task dependencies and to determine the critical path of development activities. Such an analysis is usually needed to define a realistic schedule. It is also useful in evaluating contingencies and creating contingency plans.

Change Control - This is the core of configuration management. It controls all changes to baselined products. The approval process for changes might be as follows:

- The written request for change is submitted to the configuration management function. It might come from a change in requirements or from a trouble report documenting a defect.

- An assessment is made of the technical feasibility of the change, and its impact on schedule and budget. If it has the potential to endanger life and property, a separate safety assessment may be made.

- The change is approved or disapproved based on its potential effect upon safety, its value and its cost.

- The development plan is modified and resources adjusted to add approved changes.

- The fully verified change is entered into the new baseline.

2.6.2 Requirements on the Support Environment

The activity of management imposes the following requirements on the support environment data base.

Baselined Products - The development plan, although not a part of the software system or its descriptive information, should be maintained as a baselined product to insure proper management of changes to the plan. Configuration management data and quality assurance plans should also be baselined.

Non-Baselined Data - Significant amounts of information associated with the management must be kept temporarily. This information includes engineering change requests, trouble reports, resource allocation plans, actual resource utilization reports, technical milestone status, action item status, and the results of quality assurance reviews.

Measurement Data - Many measurements are of interest to management. These include the number of engineering change proposals (ECP), and trouble reports (TR), time to process an ECP or TR, resource use for each ECP or TR, resource use by project activity, and software size and complexity measures.

3.0 Tools to Extend the MAPSE

Data Entry

Problem Expression Editor (for requirements analysis, specification)
Syntax/Template Directed Editor
Menu Manager
Graphics Package (GKS, 2D, 3D)
Word Processing integrated with Graphics and Electronic Mail
Network communications across hosts and targets

Library Aids

Semantics Information Browser
Reuseable Components Browser
Dictionary and Schema Tools
Diana Tree Browser
Host CLP script Manager

Management Aids

Report Generator
(Integrated Text and Graphics Forms Generator)
Automated Precedence Network
Automated Work Breakdown Structure
Schedule Generator
Change Request Tracker
Resource Scheduling Aid
Event Flags&Signals Generator
(signal path planning)

Syntax/Semantics Analysis

Requirements Language Processor
Requirements Information Analyzer
Design Specification Language Processor
PDL Syntax Analyzer
Design Complexity/Metrics Analyzer
Consistency/Completeness Checker
Standards Checker
Requirements to Design Tracer/Checker

Proof/Assertion Checker

Verifyer/Assertion Analyzer
Theorem Prover
Symbolic Execution System

Implementation Support

Compilation Order Analysis
Automated Recompilation
Elaboration Dependencies Analyzer
Change Control and Impact Assessment
Generic Usage Report Generator
Call Tree Report Generator
Performance Metrics Analyzer
Cross-Reference Generator
Statement Profile Generator
Diana Tree Expander

Test Generation, Analysis, Automation

Test Harness
Generic Instantiation Harness
Test Data Generator
Black Box Test Generator
Data Extraction and Reduction
Test Results Comparator
Target System Testbed (fully instrumented)
Environment Simulator/Stimulator
Performance Monitor

PDL Interpreter

Test Coverage Analyzer
Test Completeness/Consistency Analyzer

Target Emulation/Simulation
Scenario Generator
Fault Stimulator/Analyzer

Modeling/Simulation

Resource Estimator
Modeling Tool
Prototyping/Simulation Capability

Performance Model
Reliability Model

Run-Time System Support

Runtime Support Dependencies Analyzer
System Timing Analyzer
System Tasking Analyzer

Runtime Monitor

System Storage Analyzer

Distributed Target System Support

Target Node Resources Editor
Target Network Topology Editor
Partitioning and Allocation Editor
Distributed System Generator (program builder)

Distributed workload simulator

Expert Systems

Real-Time Assistant
Fault-Tolerance Assistant
Reuseable Components Assistant
Upgrade Load, Test and Integration Planning Aid (for non-stop nodes)

Expert System Generator

N89-16373

Ada Structure Design Language ASDL

Lutfi Chedrawi, M.S.
Applied Technology Division
System Engineering Department

Computer Sciences Corporation

Abstract

An artist acquires all the necessary tools before painting a scene. In the same analogy, a software engineer needs the necessary tools to provide his/her design with the proper means for implementation. Ada provides these tools. Yet, as an artist's painting needs a brochure to accompany it for further explanation of the scene, an Ada design also needs a document along with it to show the design in its detailed structure and hierarchial order.

Ada could be self-explanatory in small programs not exceeding fifty lines of code in length. But, in a large environment, ranging from thousands of lines and above, Ada programs need to be well documented to be preserved and maintained. The language used to specify an Ada document is called Ada Structure Design Language (ASDL). This language sets some rules to help derive a well formatted Ada detailed design document. The rules are defined to meet the needs of a project manager, a maintenance team, a programmer and a system designer. This paper will explain in detail the design document templates, the document extractor, and the rules set forth by the Ada Structure Design Language.

1.0 Introduction

Ada covers the different scopes under the software engineering spectrum. The Ada scopes can range from real time systems, scientific applications and other known software applications to abstract problems mapping, object oriented programming and new software engineering concepts.

Keeping this in mind, Ada can become very complicated when designing large projects governed by many different tasks, generic entities and overloading mechanisms. Therefore, design documents are needed to clarify some of the obscurities that might arise when designing large systems. The design document should also accomodate for the tools provided by Ada and support the Ada language by showing the program, entities, and tasks at the functional level. The design method, called Ada Structure Design Language (ASDL), approaches Ada from two different levels :

o The specification level.
o The functional level.

ORIGINAL PAGE IS
OF POOR QUALITY

```
-- <@@@>

-- **********************************************************************
-- *                     Package TASKS_INPUT_QUEUE                      *
-- **********************************************************************

-- <@@@>

-- **********************************************************************
-- *                                                                    *
-- * Author      :   Lutfi Chedrawi                                     *
-- * Company     :   CSC                                                *
-- * Job order   :                                                      *
-- * Contract ● :                                                       *
-- *                                                                    *
-- **********************************************************************

with UNCHECKED_DEALLOCATION;

generic
  type OBJECT is private;

package TASKS_INPUT_QUEUE is

  -- <@@@>
  --=================================================================
  --                          PACKAGE OVERVIEW
  --=================================================================
  -- Package overview ----------------------------------------------
  -- ----------------
  -- The tasks input queue will manage the input queue for a given
  -- process. The input queue manager task in this package will either
  -- put or take an object on or off the queue repectively. This task
  -- also manages the queue by adding or releasing nodes to and from the
  -- input queue and keeping a record of the queue size.
  -------------------------------------------------------------------
  --
  --=================================================================
  --                          INTERNAL ROUTINES
  --=================================================================
  -- Internal routines -- Description -----------------------------
  -- -----------------    -----------
  -- Task
  -- QUEUE_MANAGER        To manage the queue size, the reads and wr tes
  --                      from and to the input queue. In the read
  --                      operation, if no objects were on the queue
  --                      the queue manager requester will wait until
  --                      an object is put on the queue.
  -- Procedures
  -- GET                  Performs a rendezvous with the get entry in
  --                      the queue manager task to get an object off
  --                      the queue.
  -- PUT                  Performs a rendezvous with the put entry in
  --                      the queue manager task to put an object on
  --                      the queue.
  -- QUEUE_SIZE           Performs a rendezvous with the queue size
  --                      entry in the queue manager task to get the
  --                      size of the queue in terms of number of nodes
  --                      in the queue.
  -------------------------------------------------------------------
  --
  --=================================================================
  --                             EXCEPTIONS
  --=================================================================
  -- Exceptions --------- Description -----------------------------
  -- ----------           -----------
  -- N/A
  -------------------------------------------------------------------
  --
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
--===================================================================
--                         EXTERNAL REFERENCES
--===================================================================
-- External entities -- Description ---------------------------------
-- -----------------    -----------
-- N/A.
---------------------------------------------------------------------
--
--===================================================================
--                         EXTERNAL ROUTINES
--===================================================================
-- External routines -- Description ---------------------------------
-- -----------------    ------------
-- Procedure
-- UNCHECKED
--    _DEALLOCATION       This procedure will release the input queue
--                        nodes back to heap storage.
---------------------------------------------------------------------
--
--===================================================================
--                          CHANGE HISTORY
--===================================================================
-- Date/Authors ------- Description ---------------------------------
-- ------------        -----------
-- 3/31/86 - Lutfi      Added a get procedure to rendezvous with the
--                      get entry in the input queue manager task.
-- 4/1/86  - Lutfi      Added a put procedure to rendezvous with the
--                      put entry in the input queue manager task.
-- 4/1/86  - Lutfi      Added a queue size procedure to rendezvous
--                      with the queue size entry in the input queue
--                      manager task.
---------------------------------------------------------------------
--===================================================================
--                          DEPENDENCY TREE
--===================================================================
-- Dependency tree --------------------------------------------------
-- ---------------
--    Task                 Calling procedures
--    QUEUE_MANAGER --
--                    |
--    Entries         |
--       o GET        |__ GET
--       o PUT        |
--       o QUEUE_SIZE |
--                    |__ PUT
--                    |
--                    |
--                    |__ QUEUE_SIZE
--
---------------------------------------------------------------------
-- <@@@>

-- QUEUE --> to queue the request for a task in a FIFO queue.

type QUEUE;
type QUEUE_ACCESS is access QUEUE;
type QUEUE is
  record
    FORWARD : QUEUE_ACCESS ;
    BACKWARD: QUEUE_ACCESS ;
    ELEMENT : OBJECT       ;
  end record;

-- Internal routines GET, PUT and QUEUE_SIZE
--
 procedure GET        (ELEMENT       : out OBJECT  );
 procedure PUT        (ELEMENT       : in  OBJECT  );
 procedure QUEUE_SIZE (SIZE_OF_QUEUE : out INTEGER );

end TASKS_INPUT_QUEUE;
```

Example -1- : Ada specification level design document using ASDL.

2.0 ASDL levels:

2.1 - The specification level:

ASDL, at this level, will suffice the specification definition and description of a system and put the following at hands :

o Requirements	: statement definition of the overall structure.
o Author & History updates	: a log file of updates will provide another programmer and the system manager with history information of all changes made. This pin-points the responsible person for the changes made and keep track of program progression. Tracking responsibilities is needed by the system manager in case any ambiguities ever arise that need further explanation or further documentation to help clarify the changes made.
o Independability	: interfaces and hierarchies definitions of each entity.
o Maintainability	: the system will be easily maintained throughout its life cycle. ASDL will provide all the clues for a maintenance team to keep track of the environment.

2.2 The functional level :

ASDL, at this level, provides programmers with tools for debugging ease and managers with prospects on design clarity . It allows, the documentation of :

o Requirements	: statement definition of an entity.
o Structured analysis	: explaining the input/output and specifications of each entity.
o Structured design	: defining the functional flow of each entity.

3.0 ASDL format

3.1 ASDL specification level format:

ASDL will show the declaration of an entity. The data structure, functions, procedures, tasks, and packages are explained at this level in a general form without going into details. The specification level design document using ASDL is shown in example -1-. A further investigation of this example allows us to identify different entries within the specification level format. Each entry permits the documentation of a part of the system that meets the needs of the different classes of people involved. All entries are mapped to a static form which allows the derivation of a specification level template. The template skeleton is static on the outside, but the explanation within each entry can be dynamically filled with information to preserve the creator's integrity to express his own design documents.

The template entries for the specification level format serve as a road map to each or all individuals involved in the design of the entity. ASDL specification template format holds the following entries:

- o entity overview
- o internal routines
- o exceptions
- o external references
- o external routines
- o change history
- o dependency tree

The entity overview entry identifies the function of an entity. This entry serves all the classes of people involved in the development of the project. The information covered in this entry should hold the important features governining an entity. Not only would this entry serve as information coverage of the entity but also acts as a fast index to the contents of the entity under development or investigation.

The internal routines entry covers the naming definitions and the entity internal routines descriptions. A maintenance team can make use of this entry by utilizing the explanation provided to understand the problem statement definition and to identify the internal routines. Both, the project developers and system maintainers hold the responsibility of keeping the information within this entry up to date.

The exceptions, external references and external routines entries exclusively permit the system designers and project managers to recognize the system exceptions handling mechanism and to understand the system components interaction. The system designers can keep a close watch of the system by making sure that all errors are handled and a safe passage is assured by the exceptions handlers. Moreover, the exception entry will provide a fast summary of all exceptions occuring within an entity. In the same manner, the external references and routine entries will allow the project managers to check the entity interaction at both the general (External references) and specific levels (external references'internal routines.)

The history changes entry, allows the system maintainers to log all the changes made to the entity throughout its life cycle. Moreover the system developers can communicate among each other by notifying through this entry other team members of important changes.

The tree dependency nicely shows in a graphic form the entity internal hierarchy. This entry is intended to serve all the people involved in the project.

```
package body TASKS_INPUT_QUEUE is

  procedure DEALLOCATE is new UNCHEKED_DEALLOCATION (QUEUE, QUEUE_ACCESS);

  -- Input queue manager --> will retreive information from the input
  --                          queue

  task type INPUT_QUEUE_MANAGER is
    entry PUT        (ELEMENT       : in  OBJECT  );
    entry GET        (ELEMENT       : out OBJECT  );
    entry QUEUE_SIZE (SIZE_OF_QUEUE : out INTEGER );
  end;

  QUEUE_MANAGER : INPUT_QUEUE_MANAGER          ;

pragma PAGE;

-- <@@@>

  task body INPUT_QUEUE_MANAGER is

    -- SIZE -> to return the size of the input queue in terms of number
    --         of nodes in the queue.

    SIZE               : natural := 0;


    -- Function   : to manage the input queue. It either puts on or takes
    --              an object off the queue. The GET entry to get an object
    --              off the queue is guarded so the task will make the
    --              requestor wait until an object is put on the queue.
    --              This task will manage the queue size and return its
    --              value when requested.

    -- In         : N/A

    -- In Out     : N/A

    -- Out        : N/A

    --=================================================================
    --                              ALGORITHM
    --=================================================================
    -- Algorithm :
    --   begin
    --     loop
    --       select
    --         when size is greater than zero.
    --           accept get (element : out object) do
    --             get out object from the node at the forward link of head
    --               of the queue
    --             set the forward link of the node at the get node's backward
    --               link to the forward link of the get node.
    --             set the backward link of the node at the get node's forward
    --               link to the backward link of the get node.
    --             release get node back to heap storage
    --             decrement size
    --           end get;
    --       or
    --         accept put (element : in object) do
    --           create new node
    --           insert "in" object
    --           set new node forward link to head of the queue
    --           set new node backward link to head backward link
    --           set forward link of the node at the head backward
    --             link to new node
    --           set backward link of head to new node
    --           increment size
    --         end put;
    --       or
    --         accept queue_size (size_of_queue : out integer) do
    --           set size_of_queue to size
    --         end queue_size;
    --       end select;
    --     end loop;
    --   end input queue manager;
    --=================================================================
    -- <@@@>
```

```
pragma PAGE;

-- <@@@>

procedure GET          ( ELEMENT             : out OBJECT         ) is

-- Function  : to call the input queue manager task so it can get an
--             object off the queue.

-- In        : N/A

-- In Out    : N/A

-- Out       : element -> the object to be returned from the queue.

--=========================================================================
--                               ALGORITHM
--=========================================================================
-- Algorithm :
--   begin
--     rendezvous with the input queue manager task to get the object
--   end;
--=========================================================================
-- <@@@>

pragma PAGE;

-- <@@@>

procedure PUT          ( ELEMENT             : in  OBJECT         ) is

-- Function  : to call the input queue manager task so it can put an
--             object on the queue.

-- In        : element -> the object to be put on the queue.

-- In Out    : N/A

-- Out       : N/A

--=========================================================================
--                               ALGORITHM
--=========================================================================
-- Algorithm :
--   begin
--     rendezvous with the input queue manager task to put the object
--   end;
--=========================================================================
-- <@@@>

pragma PAGE;

-- <@@@>

procedure QUEUE_SIZE   ( SIZE_OF_QUEUE       : out INTEGER        ) is

-- Function  : to call the input queue manager task so it can get the
--             size of the queue.

-- In        : N/A

-- In Out    : N/A

-- Out       : size of queue -> the queue size to be returned from the
--                              input queue manager task.

--=========================================================================
--                               ALGORITHM
--=========================================================================
-- Algorithm :
--   begin
--     rendezvous with the input queue manager task to get the queue
--       size
--   end;
--=========================================================================
-- <@@@>

end TASKS_INPUT_QUEUE;
```

Example -2- : Ada functional level design document using ASDL.

3.2 ASDL functional level format:

ASDL will describe the system in a more detailed functional flow. ASDL will require system developers to combine Ada keywords and the english language to bring about a detailed flow of the entity, yet not cryptic to the designers or software maintainers. The system manager can also check the system logic and design structure for ambiguity, clarity, performance and possible implementation (i.e whether the entity can be implemented as described or whether the implementation is not possible due to misinterpretation of problem definition, requirements need, Ada weaknesses, etc....).

ASDL functional level format holds four entries as shown in example -2-. The "in", "in out" and "out" entries correspond to Ada parameters passing descriptions. The inclusion of these entries will entitle the system maintainers, developers, designers and managers to understand the input/output of system components. Moreover, a functional flow design is given by the algorithm entry to show the structure in its more detailed english like design.

Finally, ASDL tends to be similar to PDL (Process Design Language) at this level, which proves to be advantageous since no training is needed for individuals already familiar with PDL.

4.0 ASDL rules

4.1 ASDL specification level rules:

In general, ASDL does not impose any rigid rules. The ASDL rules for the specification level format should insure the derivation of a design document. The rules are set to give a detailed explanation of entities interactions, entity specification and data representation. The specification level format can be mapped onto the following rules :

- o new updates should be entered when necessary
- o dependency tree should be leveled to show the new entities in their hierarchial depth
- o history logs updates should cover the changes made
- o the information should be entered under the specified field to insure the extractor ability to perform its functions.

4.2 ASDL functional level rules :

At this level, ASDL requires developers to respect the outer and inner structures of Ada blocks, statements and looping mechanisms. The Ada keywords should be entered to show the Ada flow as if it was coded. Moreover, the Ada keywords should be combined with a detailed explanation in english to show the flow of Ada statements. The advantages of combining Ada keywords and english words will divulge when the implementation phase takes place. The project's implementation will become a matter of mapping the algorithm to Ada code.

5.0 ASDL extractor

5.1 The formatter extractor:

ASDL extractor is envisioned as a formatter extractor with menu driven options. When asked to format a document written by ASDL, the extractor will prompt the requestor with a menu. The menu selection can be accessed through cursor control.

ASDL formatter extractor should come with default values to allow simple extracting and echo printing of text to the specified destination file. On the other hand, if required, the extractor should perform all the necessary tasks to derive a well formatted Ada design document including centering of titles, margin justification, page numbering and other functions found in wordprocessors. Moreover, the extractor can control the part of text to be extracted from the document by a simple turn on/off flags or toggle keys (if an interactive session is requested.) Those flags are shown in example 1 & 2 as '@', '#' signs preceded and succeded by '<' and '>' designators.

6.0 Conclusion

In summary, ASDL will prolonge the software life cycle. In addition, it will allow the documentation of large systems otherwise might become very difficult to understand. Finally, ASDL will act as a communicae to all the classes of individuals involved in the system development.

---

* Ada is a registered trademark of the U.S government, Ada Joint Program office.

* ASDL extractor is still under development.

REFERENCES

[1] Barns, J. G. P. Programming In Ada. Addison-Wesley Publishing Company. Second Ed.

[2] Booch, G. Software Engineering With Ada. Benjamin/Cummings.

[3] Caine S. and Kent G. PDL - A Tool For Software Design. IEEE Computer Society Press.

[4] Linger C., Mills H. and Witt B. Structured Programming. Addison-Wesley Publishing Company.

[5] Privitera J. P. Ada Design For The Structured Design Methodology. IEEE Computer Society Press.

[6] ---, Reference Manual for the Ada programming Language. Ada Joint Program Office, Department of Defense.

[7] Zelkowitz M., Shaw A. and Gannon J. Principles Of Software Engineering Design. Prentice-Hall, Inc.

N89-16374

Artificial Intelligence & the Space Station Software Support Environment

Gilbert Marlowe

In a software system the size of the Space Station Software Support Environment (SSE), no one software development or implementation methodology is presently powerful enough to provide safe, reliable, maintainable, cost-effective real-time or near real-time software. In an environment that must survive one of the harshest and lengthiest lifetimes, software must be produced that will perform as predicted, from the first time it is executed to the last. Many of the software challenges that will be faced will require strategies borrowed from "Artificial Intelligence (AI)." In the Statement of Work (SOW) for the SSE, AI is the only development area mentioned as an example of a legitimate reason for a waiver from the overall requirement to use the Ada™ programming language for software development. While it

Ada™ is a Registered Trademark of the United States Government, Ada Joint Program Office (AJPO)

is recognized that some solutions are not readily amenable to solution in contemporary Ada Programming Support Environments (APSEs), it is clearly the intent of the SOW that there be one development language for all of Space Station software so that configuration management, system definition and reuse of verified and validated software be as simple and as efficient as possible. This paper will attempt to define the limits of the applicability of the Ada language, APSEs (of which the SSE will be a special case), and software engineering to AI solutions by describing a scenario that involves many facets of AI methodologies.

The scenario itself is fairly simple. It involves the Space Station, an undocked Space Shuttle, and a robot unattached to either the Space Station or the nearby Shuttle (the robot is equipped with vision sensors, a propulsion system with translational and rotational jets, and manipulators/grapplers). The robot will start in proximity to the Station either stationkeeping or performing a low priority task that may be preempted. At the request of one of the specialists onboard the Station the robot begins to maneuver itself to the nearby Shuttle. If the Shuttle is near enough, the robot will be guided by the Station Trajectory Control Program. If the Shuttle is any appreciable distance away the robot will request Guidance, Navigation and Control (GN&C) programs necessary to compute and maintain a trajectory to the Shuttle. It may also request the Station Trajectory Control Program to calculate intermediate vectors that it will use to compare against during the rendezvous. While all of this was happening, the specialist onboard the Station identified and requested a software load in addition to the GN&C software being loaded by the robot. This software included a vision system, general GN&C programs to be used in proximity operations at the end of the rendezvous (this could be detailed enough to allow the robot to literally settle down in a specified position and attitude in the cargo deck of the specified Shuttle without any human intervention, or it might allow a specialist onboard the Shuttle to interactively guide the robot to the desired location and attitude) and a task identification that will establish whether or not this task may be preempted and, if it can, by what other tasks or levels of tasks. Once the necessary software has been loaded, the robot is essentially a free agent and must vie with other agents for Station computing resources. As soon as it begins an escape trajectory, the robot begins to interface with the Station Collision Avoidance Program (CAP) to establish and maintain a clear trajectory. Very likely the robot and the Station will enter a dialogue, with the robot proposing a trajectory and the CAP either accepting the proposed trajectory or denying it. If the trajectory is denied, it is the responsibility of the robot to calculate another trajectory, using Station computing facilities if necessary. This cycle of calculation, proposal, and verification will proceed until an acceptable trajectory is proposed, acceptable meaning that the proposed trajectory does not involve undue risk of collision between the robot and the Station or the robot and other free flyers, and that the proposed trajectory is reasonable given the amount of propulsive and non-propulsive consumables that have been budgeted for this task (a configuration item that will be maintained by the Station Object Base). The robot is responsible for calculating a trajectory that meets the specified goals :

that the rendezvous occur within a specified amount of time, that the rendezvous cost no more than a specified amount of non-reusable resources, and that the rendezvous occur with a specified object (rather than that the rendezvous occur at a specified place). The Station maintains configuration control over trajectories using the CAP and will not allow trajectories that violate safety standards.

After the robot has negotiated a safe trajectory, it still must maintain a dialogue with the Station so that both are aware of the robot's current and predicted position in any given time quanta. This dialogue is necessary to keep the CAP current and so that the robot may be informed of any changes in the trajectory or in the task.

When the robot arrives in near proximity to the Shuttle it has been assigned to rendezvous with, it will announce itself to the Shuttle computers. At this point, depending on the software loaded at the Station, the robot may or may not be able to proceed to dock without any human intervention. If it is capable, the robot will inform the

Shuttle computers, and begin a docking sequence. At any time the humans onboard may elect to override the automatic docking sequence and control the robot through their onboard computers. If the robot has not been loaded with the appropriate software, it will announce this and wait for further instructions. Shuttle specialists may decide to either load the software necessary for an automatic docking sequence into the robot, or manually control the docking sequence.

Once the robot is securely docked, a specialist in the cargo bay begins refurbishment and outfitting of the robot. The old manipulators/grapplers are removed and new ones are attached. The robot is refilled with consumables for the next segment of its task and, in parallel with all of this activity, new software is loaded into the robot. This new software will guide the robot to a satellite at a geosynchronous altitude, direct the robot to grapple the satellite (which will require the robot to make contact with the satellite in a very specific attitude with very specific rotational and translational velocities as well as a sequence of grapple maneuvers that must be performed as directed to ensure stability), and return to the Shuttle or to the Station so that the satellite may be repaired. Alternatively, if the repair is simple enough (such as increasing the spin of the satellite) the robot may perform the indicated repair and return to the Station (if supplies of consumables allow return to the Station rather than refueling at the Shuttle). The close in proximity operations immediately preceding the grapple will require a number of real-time computations. The robot must visually confirm that the satellite is the correct one, that the approach is proceeding nominally, and that grapples are being manipulated in the correct sequence and towards the correct targets on the satellite. Trajectory programs in the robot must calculate burns that will match translational and rotational velocities of the two vehicles and manipulator control programs must monitor and guide grapplers from an unsteady platform toward targets that are moving. As soon as the manipulaotr control program confirms that the satellite has been securely grappled, the robot begins to contact the Station. It reports the successful completion of rendezvous, approach and grapple and again enters negotiation with the CAP, this time for a return trajectory. When a acceptable trajectory (which will be based on new mass properties and different consumables loadings that reflect the current robot/satellite pair's configuration characteristics) has been agreed to by both parties, the robot will begin its trip home to the Station. As before, the robot will maintain contact with the CAP and perform maneuvers as required or as requested by the CAP until it is docked at the Station.

This scenario illustrates the flexibiltiy offered by allowing a general-purpose robot to serve as an free agent to perform a task that would be uneconomical if performed by humans or if performed by a robot that could not perform unless guided by humans or Shuttle or Station computers. A robot may be treated as an agent and allowed to compete with other agents for computing and other shareable resources to maximize the efficient use of those resources. Obviously computing time and consumables will both be at a premium for the

Station since neither is a renewable resource. Just as obviously, it is more efficient to send a robot to do many tasks rather than sending a manned vehicle with the life-support system that it must provide. An added benefit to treating the robot as a separate agent is that in the event of a communication failure the robot would be able to continue the task until such time as communications are restored or the task requires communication (such as the negotiation for trajectories described before). This also makes efficient use of human resources and offloads computing work to the responsible agent - the robot. A subtle, but important, benefit is that this approach separates the specialists from details about how the robot fullfills the task assigned to it (similar to the way that object-oriented design hides implementation details from the user) allowing him/her to worry about the overall task rather than details that are subject to change dynamically (such as a trajectory that fullfills the task requirement without violating Station safety constraints).

All of the software discussed in this paper should be implemented in

Ada™ to ensure consistancy of interface between the software modules. The Ada™ construct of packages will allow software to be developed in modules that are additive to the total software functionality. The time is now to start deciding not _whether_ Ada™ should be used for AI applications on the Station, but _how_ to efficiently use the power of Ada™ to develop software modules that are sufficiently well engineered to meet real-time requirements in problem spaces that may not allow a complete description at any given time. To introduce another language on Station doubles the complexity of configuration management. To introduce another language on Station that cannot support strong typing will double again the configuration management task. It is clear that Ada™ is _sufficient_ for many applications in AI, but it is not clear that another language is _necessary_ for AI applications or that a trade off between power in expressing a solution using a "traditional" AI language (i.e. Lisp, Prolog) and the resources required to maintain any type of configuration control (including verification, validation, testing and safety data) over a configuration item produced using that language is worth the price. Perhaps it is too early to tell, but it is my hop that by discussing now what the Station will require in the future we may have a crystal vision of our intermediate and long-term goals and the tools we will use to reach those goals. I think that discussing scenarios such as the one above will prove fruitfull in determining the direction that the Space Station SSE will take.

Gilbert Marlowe
c/o Rockwell Shuttle Operations Company (RSOC)
600 Gemini Blvd.
Houston, Texas 77058
(713) 282-2760

NASA Mail code : RS16

Acknowledgements :

Special thanks to Dr. Charles W. McKay for his support and guidance.

Thanks also to Lisa Willingham for moral support and the abilty to listen to me ramble on long after a mere mortal would have gone to sleep.

N89-16375

SPACE STATION ADA RUNTIME SUPPORT FOR NESTED ATOMIC TRANSACTIONS

Edward J. Monteiro
McDonnell Douglas Astronautics Co. - Houston
16055 Space Center Blvd.
Houston, Texas 77062
(713) 280-1629

The Space Station Data Management System (DMS), associated computing subsystems, and applications have varying degrees of reliability associated with their operation. On one hand, payload applications and associated processing can fail or have interrupted service without endangering the operation or safety of the Station. On the other hand, subsystems such as the Environmental Control and Life Support Subsystem (ECLSS) must be fail safe. The Guidance and Control subsystem must be non-stop in nature. These different levels of reliability require corresponding levels of support from the runtime environment.

A model has been developed [McKay 86] which allows the DMS runtime environment to appear as an Ada virtual machine to applications executing within it. This model is modular, flexible, and dynamically configurable to allow for evolution and growth over time.

Support for Fault-tolerant computing is included within this model. The basic primitive involved in this support is based on atomic actions [Grey 78, Lampson 81]. An atomic action posesses two fundamental properties: 1) It is indivisible with respect to concurrent actions and 2) it is indivisible with respect to failure. These properties allow rollback and recovery to occur in systems which encounter erroneoneous computing. Using this primitive as a building block, higher levels of fault-tolerant support can be achieved. A transaction is a collection of atomic actions which collectively appear to be one action. Transactions may be nested, providing even more powerful support for reliability. Transactions and nested transactions exhibit the same fundamental properties as an atomic action. This abstraction has found widespread usage in database technology and non-stop computing environments [Meuller 84, Comm 85].

This paper describes a proposed approach to providing support for nested atomic transactions within the Ada runtime model developed for the Space Station environment. The level of support is modular, flexible and dynamically configurable just like the overall runtime support environment. These characteristics of the model allow for the varying needs of reliability to be met under conditions where fault avoidance cannot be guaranteed. The paper discusses:

a) the requirements which must be addressed in the Space Station DMS environment.

b) an overview of the transaction model

c) the proposed protocol model

d) an example of its use

e) the Ada syntax and semantics associated with the protocol

f) and, other modules needed to complement the transaction model

N89-16376

REUSABLE SOFTWARE PARTS AND THE SEMI-ABSTRACT DATA TYPE

Sanford G. Cohen
McDonnell Douglas Astronautics Company
P.O. Box 516
St. Louis, Mo. 63166

The development of reusable software parts has been an area of intense discussion within the software engineering community for many years. More recently, Ada has been promoted as having the facilities for developing reusable software. However, there have been few attempts to validate reusability concepts in practice for real-time embedded applications such as missile navigation, guidance and control. For these applications parts must not only be reusable, but they must also be efficient and easy to use.

Missile guidance, navigation and control applications are noted for severe constraints in terms of processor size and computational requirements. In this paper, the author describes an approach for developing reusable parts for these applications which meet the following reusability criteria:

1. Reusable: Capable of being used/reused in a wide spectrum of applications within the domain for which the parts were developed.

2. Tailorable: Capable of being customized to the precise requirements of the using/reusing applications.

3. Efficient: Capable of operating within an environment which is severely constrained in terms of both memory and execution cycles.

4. Simple to Use: Capable of being effectively used by the average software engineer.

5. Protected Against Misuse: Capable of detecting obvious misuse.

Validating the feasibility of developing reusable parts which possess these characteristics is the basis of the Commom Ada Missile Packages Program (CAMP), an Air Force sponsored program under contract tot he McDonnell Douglas Astronautics Co. - St. Louis. Under CAMP, over 200 reusable software parts have been developed, including parts for navigation, Kalman Filter, signal processing and autopilot. This paper is an outgrowth of work done on that project.

The author presents six different methods for designing reusable software parts. (These methods are illustrated in the accompanying figure.) The author examines these methods through determining the impact of each method on developing a single part. He compares the methods against four evaluation criteria:

1. Appropriateness of the interface

2. Control for preventing misuse

3. Availability of needed mathematical operators and functions

4. Degree to which user's job is simplified

Each of these criteria is essential for developing parts which can be used, are reusable and are sufficiently efficient for missile navigation, guidance and control applications.

The author proposes the use of a generic approach, called the "Semi-Abstract Data Type" method, for developing reusable parts and provides a rationale for this selection. The semi-abstract data type method makes full use of Ada's generic and strong data typing facilities to create parts which are reusable, tailorable, simple to use and protected from misuse. The method achieves efficiency through the choice of data structures which are compatible with efficient algorithms and through implicit definition of user data structures.

Informal Report by the ARTEWG*

Mike Kamrad
Honeywell Systems and Research Center
Minneapolis, Minnesota

*ARTEWG - Ada Run-Time Environment Working Group

This session will provide a status report and an update on the ARTEWG activities.

SESSION G.4 COMPUTERS FOR ADA

Session Chair: Charlie Randall
GHG Corp.
Houston, TX
and
Rod Bown
University of Houston - Clear Lake

The following presentations are informal.

G.4.1	Language Directed Machine Lawrence Greenspan Ronald Singletary Sanders Associates Nashua, New Hampshire
G.4.2	Ada Port to the ELXSI System Ralph Merkle ELXSI San Jose, California
G.4.3	Message Passing Concurrent Processing Architecture Tony Anderson Intel Scientific Computers Beaverton, Oregon

SESSION G.5 — DIALOG WITH THE NASA SOFTWARE WORKING GROUP

Session Chair:	Robert Nelson Goddard SFC and helper Richard Kessinger SOFTECH

This session provides an opportunity for the NASA Software Working to identify their existence, role and scope. The audience is invited to participate in a question/answer session.

CONCLUDING REMARKS

Jack Garman, NASA Lyndon B. Johnson Space Center
and
Charles W. McKay, University of Houston-Clear Lake

NASA-JSC