1987

NASA/ASEE SUMMER FACULTY RESEARCH FELLOWSHIP PROGRAM

MARSHALL SPACE FLIGHT CENTER
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

CAN SPACE STATION SOFTWARE BE SPECIFIED THROUGH ADA?

Prepared By:	Arthur Knoebel
Academic Rank:	Professor
University and Department:	New Mexico State University Mathematical Sciences
NASA/MSFC:	
Laboratory:	Information and Electronic Systems
Division:	Software and Data Management
Branch:	Systems Software
NASA Colleagues	John W. Wolfsberger Robert L. Stevens
Date:	August 20, 1987
Contract No.:	The University of Alabama in Huntsville NGT-01-008-021

ABSTRACT

Programming of the Space Station is to be done in Ada. A breadboard of selected parts of the work package for Marshall Space Flight Center is to be built, and programming this small part will be a good testing ground for Ada. One coding of the upper levels of the design brings out several problems with top-down design when it is to be carried out strictly within the language. Ada is evaluated on the basis of this experience, and the points raise are compared with other people's experience as related in the literature. Rapid prototyping is another approach to the initial programming; several different kinds of prototypes are discussed, and compared with the art of specification. Some solutions are proposed and a number of recommendations presented.

ACKNOWLEDGEMENTS

Many thanks are due many people. Without attempting to mention everyone, I will simply express my gratitude to John Wolfsberger, Robert Stevens and David Aichele for making it possible for me to be here another summer, and to Ellen Williams and Catherine White for their help in learning to use the computers in the Language Laboratory.

TABLE OF CONTENTS

I. INTRODUCTION

The Problem. The Space Station is to be programmed in Ada. How well will this relatively new and untried language fare? Selected parts of the Space Station are to built on a breadboard. Now may be the time to start programming this model to see how well Ada will work out. In particular, can Ada be used to specify or prototype the software? If not, when should Ada be introduced into the life cycle?

Results. Separate compilation of the specifications and bodies of subprograms in Ada makes possible top-down design of the Space Station software. However, one is not free to cut off the coding for a procedure or the declaration of a data type indiscriminately. Thus to go a ways in the coding, one needs to know something of the configuration of the computers on which the software is to run. Moreover, one needs to know something of the data flow and the nature of the data types to be used for input and output. Since this information was not available, only a little bit of code could be produced. Crucial to obtaining good coding is knowing when to start programming on such a large project. From this viewpoint this summer project is premature.

The Ada library, into which compiled units go, has no explicit structure. There are implicit dependencies of one unit on others, but the programmer needs help from the software development environment to keep all this straight.

Connected with this is what style of programming should be used: should it be hierarchical, with a deep tree structure, or should it be like an alphabet soup with a large number of separate tasks and subprograms?

The fundamental recommendation is to defer coding in Ada until after the traditional techniques of specification and design have gotten the software organized.

Overview. We review the background of the Space Station, where the project is now, and the role of Ada in it. Next is presented several philosophies of programming,

particularly as regards Ada. One of these is illustrated by a sample program.

Out of this we present a critique of Ada and compare our observations with those already presented in the literature. A number of solutions to problems with Ada are given.

Ada as a Stimulant. This introduction closes with a comment about this writer's experience with Ada this summer and his reaction to it. More controversy surrounds Ada than any other programming language. Much has been written about its merits and demerits, as well as more generally about what language features and combinations of them are really feasible. Quite possibly, its eventual value will be seen more in the high quality discussions and debates it has engendered and in the resulting clarification of software issues rather than in its use in coding. This leads this writer to suggest that every ten to twelve years a new truly general purpose language should be designed, building on recent software experience and on projected advances in hardware. Of necessity no one person can be an expert in all features; hence the need for a panel again to design it and achieve a consensus to ensure widespread use.

II. BACKGROUND

Space Station. President Reagan, proposed in his State of the Union message in 1984 a permanently manned earth satellite orbiting the earth. Congress approved this, phases A and B are completed, and now NASA is reviewing the proposals for phases C and D to determine which contractors will design it in detail and build it. The work is split into four work packages, each the responsibility of a separate NASA site. Marshall Space Flight Center is to oversee the Laboratory, Logistics and Habitation modules, plus related work. For these contracts, Boeing and Martin-Marietta have submitted bids.

Ada has been mandated as the programming language for the Space Station.

Request for Proposal. In the Request for Proposal for phases C and D we find the Software Requirements Specification of the Laboratory Module. To give a flavor of the detail now known in the Space Station so we can illustrate in the next section the extent to which this can be converted to Ada code, we show a small section of this document [RFP] where it outlines ECLSS, the Environmental Control and Life Support System. We quote from pp. 18-22 and indicate by ellipsis those interior portions which we are omitting.

"3.4.7 ECLSS Temperature & Humidity Control (THC)

...

3.4.8 ECLSS Atmosphere Control and Supply (ACS)

...

3.4.9 ECLSS Atmospheric Revitalization (AR)

3.4.9.1 Inputs

The ECLSS AR software shall accept the following input:

a. atmospheric makeup range limits for carbon dioxide and contaminants in the module atmosphere.

b. atmospheric makeup sensor data for carbon dioxide and contaminants in the module atmosphere.

c. AR equipment performance and status sensor data.

d. requests for subsystem initiation, control, and reconfiguration.

3.4.9.2 Processing
...

3.4.9.3 Outputs
...

3.4.10 ECLSS Fire Detection and Suppression (FDS)
...

3.4.11 ECLSS Water Recovery and Management (WRM)
..."

Breadboard. Even though the contracts for the detailed design and construction have yet to be let out, a breadboard of selected features of the Space Station is under design now in the Systems Analysis and Integration Laboratory in building 4610. The Software supporting it will be written by the Information and Electronic Systems Laboratory.

Three computers are to be used: A Microvax II for the Data Management System, a Sun Workstation for displays and user interaction, and a third for the simulator of the physical systems.

Specifying versus Prototyping. At their extremes these two very different choices for top-down design are described in the article [BGS]:

> "Specifying: Develop a requirements specification for the product. Develop a design specification to implement the requirements. Develop the code to implement the design. Again, rework the resulting product as necessary.
>
> Prototyping: Build prototype versions of parts of the product. Exercise the prototype parts to determine how best to implement the operational product. Proceed to build the operational product, and again rework it as necessary."

The authors of this stimulating article go on to describe an experiment conducted to compare these two modes of software design. They conclude that prototyping is definitely cheaper but tends to produce less functional code. In more detail, to again quote them on the relative merits, they cite these benefits of prototyping:

"products with better human-machine interfaces;
always having something that works."

Three negative effects of prototyping were:

"proportionally less effort planning and designing, and proportionally more testing and fixing;
more difficult integration due to lack of interface specifications;
a less coherent design."

See also section 7 of the paper [HI] for more trade-offs.

There are two species of prototyping: vertical and horizontal. The vertical does only selected parts of the project but does those in detail. The horizontal does something on all tasks but only crudely. It is important to know at the outset which style one wishes to follow.

The book of R. J. A. Buhr has a decription in section 1.2 of the software life cycle initiated by specification. Clearly the life cycle will be different for prototyping.

III. PROGRAMMING

Philosophies and Styles. While Ada encourages and even enforces good programming practices, there is much leeway left as to how an individual programmer may develop his coding from initial conception to finished product. Since NASA's Space Station is definitely a collective and not an individual effort, considerable attention should be paid to formulating a common style of top-down design of software which is compatible with Ada.

First consider two philosophies. The first is to make use of the facility in Ada for separate compilation. In Ada specifications and bodies of subprograms may be compiled individually as they are written. This allows the deferral of decisions while at the same time coding may be started. It also allows, to some extent, the top-down specification of the software with no modification of code already produced.

The second philosophy is to allow for modification of existing code in order to fill out packages and subprograms with tasks. This requires new compilation, not only of the particular unit being recompiled, but also of all units which depend upon it. This is one its disadvantages. The obvious advantage is that more structure can be exhibited within the code itself.

We illustrate the second style of programming extensively, and then comment on the first.

A Program. We illustrate the second philosophy by following quite closely the portion of the Request for Proposal that was excerpted in the previous chapter. Our top-most package for the Environmental Control and Life Support System is simplicity itself.

```
package ECLSS is
end ECLSS;
```

This is compilable. We introduce the components of ECLSS by adding more packages inside this one.

```
package ECLSS is

    package TRC is
```

```
      end TRC;

      package ACS is
      end ACS;

      package AR is
      end AR;

      package FDS is
      end FDS;

      .
      .
      .

end ECLSS;
```

We recompile to check syntax. By way of example on how to proceed, we will refine the package AR. As the RFP lists three parts: INPUT, PROCESSING. AND OUTPUT, these are entered as packages within AR in the obvious way.

Let's refine INPUT, ignoring the other packages. We could proceed by introducing more packages, but it is seems appropriate now to introduce tasks, since we want concurrent activity of some parts of INPUT.

```
      package AR is

            package INPUT is
                  task ATMOS_RANGE_LIMITS;
                  task ATMOS_SENSOR;
                  task STATUS;
                  task REQUESTS;
            end INPUT;

            package PROCESSING is
            end PROCESSING;

            package OUTPUT is
            end OUTPUT;

      end AR;
```

Again this compilable, i.e., syntactically correct.

The RFP goes a bit beyond this in detail, but I don't think we can refine what we already have any further without losing compilability. To see how it might look, we refine, as best we can, the task ATMOS_SENSOR.

```
task ATMOS_SENSOR is
     entry CO2_DATA;
     entry CONTAMINANT_DATA;
end ATMOS_SENSOR;

task body ATMOS_SENSOR is
     CO2: fixed
     CONTAMINANTS: ARRAY (1..S) of fixed;
begin
     select
          accept CO2_DATA
               do get (CO2);
          end CO2_DATA;
     or
          accept CONTAMINANT_DATA
               do get (CONTAMINANTS);
          end CONTAMINANT_DATA;
     end select;
end ATMOS_SENSOR;
```

At this point we begin to see some of the limitations of Ada for software specification and prototyping. We are told by the RFP that there are contaminants to worry about, but no details about what they might be, or even their number. Thus we must introduce a variable S for their number which is to be filled in later. Also we are assuming we need only one number, a component of the array CONTAMINANT_DATA, to specify the extent of a particular contaminant.

Here are some comments about the coding for tasks. Ada makes provisions for a rendezvous so that concurrently running tasks may communicate with each other. We are assuming that there are some kind of lines or other input into the central processor bringing in signals telling how much carbon dioxide there is, etc. The 'select' statement chooses between the two 'accept' statements; in what sense it alternates at random between the two depends on the particular implementation of Ada; with additional coding one can make this more precise and independent of the implementation. Finally, the command get is not standard Ada and needs to be defined further.

Another Approach. Following the first philosophy that once coded, a package or subprogram should not have to be recompiled, barring mistakes, we could rewrite the preceding code. We would need to redo it as a flat horizontal design using procedures with body stubs. The idea is to specify declarations without having to write the bodies, which will be filled in later. Since the RFP is so limited in detail,

this did not seem worthwhile to pursue. Those familiar with Ada will readily see how this can be done.

Comments. This style of programming raises a number of questions which must be answered before full scale coding is undertaking. Should one use procedures or packages? (At least one procedure is needed to start execution, according to Ada rules.) This last comment centers around the question alluded to earlier. How should the Ada library of packages and subprograms be organized and extended: by units which are compiled once and more units added on down the road, or with units that are to be continually recompiled?

Clearly many more tasks are going to have to be created to accommodate all the simultaneous sensing, controlling and potential alarming that must be done. But before this can be done, we need to know the configuration of computers and the processes to be run on each.

Now this configuration may well be specified by data flow diagrams. This is something Ada does not support, and it is perhaps the most serious drawback to using Ada as a specification language. See [Buhr] pp. 83-86 and pp. 94-101 for an extended discussion of this important issue.

IV. CRITIQUE

In this chapter we address problems encountered in attempting to code immediately the specification and prototyping of the software for the breadboard. We break these up into software and hardware difficulties. First though we recall some of the strengths of Ada.

In Praise of Ada. Of the four ways to evaluate languages set forth in my earlier report [Kno1], only one, the method of qualitative matrices, has been done in depth for more than a few languages. For each of the various language features needed in the matrix for the Space Station, Ada generally does as well or better than any of the other languages surveyed.

Ada solely by itself would be hard to use. Within a good support environment it becomes a productive tool. The article by Vittorio Frigo has much praise for the VAX Ada tools written by the Digital Equipment Corporation, otherwise known as APSE, and was written after the author had written and debugged an application program. Frigo had minor complaints about the trickiness of dealing with syntax in the language-sensitive editor and the difficulty of learning the debugger. But overall he was impressed by DEC's software support.

Software Difficulties. We present four problem areas.

Separate compilation of specifications and bodies of subprograms is a powerful feature of Ada which encourages modularization. Clearly it should make possible top-down design of the software. However there are limits. In the Ada library, units can be compiled separately, and linked together according to their dependencies. Unfortunately these dependencies are not made explicit by the Ada library. A programmer must keep a separate log of these dependencies together with what has been compiled.

In Ada there are lots of data types and woe to the programmer who attempts to violate the strong typing constraints. Data has to be typed and declared to some extent in Ada. Unfortunately, many times in the initial stages of specification we would like not to do this, and instead only say that some kind of unspecified data is to be

passed. The difficulty here is that initially we may not know enough to satisfy the typing requirements of Ada. For example, contaminants are mentioned in the RFP. But not how many or how they are to be measured so that suitable ranges for their values may be specified. Thus we are stymied in our attempt to sketch out the overall structure of the software directly in Ada code.

Another nice feature of Ada is the provision for stubs in subprocedures. When a procedure Q within another procedure is incompletely known, we can simply write:

```
procedure Q is separate;
```

When finally Q is figured out, we can write the appropriate compilable package. The shortcoming of this feature is that partial information about the procedure can not be written in; we must keep it on a separate piece of paper. We can not simply work on the procedure until our fund of knowledge for it exhausted, stop and then compile. This is perhaps natural in terms of designing a workable compiler but it has the disadvantage of forcing a certain coarse granularity into the specification process.

Hardware Specifications. The last point concerns when coding should start vis-a-vis the specification and design of the hardware. Ada programming can start earlier than with most languages. But it is premature to start programming now, as was attempted in this project. To proceed further at this point we need further information on both the hardware and software to be designed. And on projects in general when should programming start? There needs to be a substantial understanding of the specific computers to be used, their configuration, the input and output to each and the data flow among them, before code can be committed to the library and hence before high-level specification can begin. Also we must decide how to organize the upper levels of the tree of packages and procedures, and how to manage the library.

V. SOLUTIONS

Recommendation. Our principal finding is that it is premature to start programming in Ada right away. We can either try some of the specification tools extending Ada, described below, or better yet employ the old fashion solution of a good English exposition of the specifications. When used correctly, concisely and accurately, our mother tongue can serve us well. Fuzzy thinking and poorly planned hardware of course will get in the way But this is not the fault of the Queen's English. As the specification moves along, gathering up more and more detail, appropriate mathematical and technical jargon should be introduced as necessary to clarify. Then program in Ada.

Specification. SSE. Lockheed has been awarded the contract to build a Software Support Environment for the Space Station. The requirements specification for this package will be available soon. There will be four subpackages (re)programmed in Ada. These package will be designed around the Apollo work stations.

This SSE will greatly affect how we proceed. In particular how will it contribute to top-down design? Into the SSE should be incorporated a scheme for managing and structuring the Ada library. Also there should be provisions for simulating the stubs in the subprograms so that the software can be run, even though it is not completed.

TAGS. Teledyne-Brown is developing the design tool called Technology for the Automatic Generation of Software. This promises to generate code automatically from detailed diagrams of data flow. It is hierarchically organized so that the design can be done top-down directly from the engineering specifications of the hardware whose software is to be coded.

We see at least three problems if this computer-aided specifier were to be used to produce Ada coding for the Space Station. First the emphasis is on detailed flow charts; but the detail may not be initially available. Also many computer scientists do not consider flow charting the best way to organize a program in order to show the tree structure of dependencies of its various parts.

Nevertheless this points up to the need for the specification of data flow at an early stage, which Ada, because of its strong typing, inhibits. Recommendation: into the SSE incorporate the specification early on of data flow.

TAGS seems to be making an end run around Ada by inventing a quite different language for the specification of the software, and only when this is completed in detail do we see Ada code generated. It would seem better to extend Ada as necessary to generate high-level specifications so that one eases naturally into the finished coding, all done in Ada.

Finally, not all features of Ada will be used in the automatic generation of the final code. This raises serious questions. Does this take full advantage of Ada? Is this really subsetting in disguise? Does it satisfy the mandate to program the Space Station in Ada?

In this connection we mention the book of Buhr, which has in chapter 3 a scheme of pictures, reminiscent of the flow charts of TAGS for notating data flow.

Prototyping Language. What is proposed here is a high-order language to be used for both specification and prototyping of software. It should be superimposed on top of Ada. Presumably it would be part of the SSE.

Anna. An example of this is the extensive project [LNR] now under way at Stanford university to develop what they call a wide spectrum language. In their words, "a wide spectrum language is a notation for describing the intended behavior of a system and the implementation of that behavior. The notation for intended behavior is usually based on a formal logic or algebra and describes what the system will do in formal terms. The implementation notation is usually concerned with efficiency of execution on hardware, and describes how the system will operate in great detail."

There are two major components of their system: Anna, a language for specifying Ada software; and TSL, Task Sequencing Language, a language for specifying distributed Ada systems. There were four principal considerations for Ada. In there words, "constructing annotations should be easy for the Ada programmer ... . Anna should provide language features that are widely used in the specification and documentation of programs. Anna should provide a

framework within which the various established theories of formally specifying and verifying programs may be applied to Ada. Annotations should be equally well suited for different applications during the life cycle of a program."

These brief excerpts do not do justice to this excellent and ambitious project; there are several parts and many more auxiliary tools not mentioned here. It is highly recommended that NASA get the latest documentation to study this system in more detail.

Algebra. There is a rigorous theory of the algebra of abstract data types, on which are based a number of languages, some already in existence and some still on paper. This is another approach to prototyping since in the algebraic theory one need not give algorithms for the operations to be performed in a procedure but, for the purposes of prototyping, one may simply give a short but complete set of properties or relationships which they must satisfy.

One such language is UMIST OBJ, outlined in the paper [GC]. As all such languages are, it is based on equational logic, given in axioms (i) to (v) of the paper. However to express the typical properties needed in computer science it is necessary to accommodate conditionals, i.e., implications and partial operations, which will encompass such things as popping empty stacks.

Unfortunately the authors of this paper seem to forget that the axioms they give must be considerably modified and extended to include these more general staements or pseudoequations. Nevertheless, it is known how to set things up to include these more general specifications (See [Kno3] and [Kno4]).

Along these same lines is the prototyping system of B. Belkhouche [Bel]. He describes a system for translating abstract data types into actual code. He was heading for code in the language PL/1 but his source output file in Appendix A has an uncanny resemblance to Ada syntax. So Ada generics could have been used here to generate compilable code better than his PL/1 code.

Other. In this last section are collected an assortment of miscellaneous suggestions for capitalizing on Ada.

Expertise. Marshall S. F. C. should develop an expertise in Ada. Several local people should learn it well and become familiar with its many facets and the literature describing and documenting the controversial issues surrounding this large and extensive language.

In this connection, an Ada library should be developed and include selected books, journals, video courses, and reports from our sister NASA sites.

Information. More information on the breadboard needs to be known. Have written down what the scope of the breadboard is to be; what it is to accomplish; and what is to be learned. Have written down the scope of the programming effort.

Miscellaneous. It almost goes without saying, do strong typing, and even make it stronger than Ada demands.

In the article [ACGE] are solutions to some common problems with Ada:

To reduce the depth of nesting, see p. 142;

Whether to decompose large programs into library units or subunits, see p. 161;

For how Ada may affect the specification phase in the life cycle, see p. 176.

In designing the breadboard, gain practice and experience in recognizing where (see [ACGE]) generics can be used to avoid duplicating common code.

Final observation. To anticipate changes in 'maintenance', modularize according to accepted concepts in the field of application.

VI. SUMMARY

This study reports on exercises done to see how well the programming language Ada supports high-level specification and prototyping. The conclusion to be drawn so far is that while Ada has a number of strong modularization features which allow for some incompleteness in coding, its strong typing prevents it from being used at the very start of a project, and most programmers will want some assistance with data flow, which Ada does not provide. The recommendation is to use English as the specification language as in the past, and perhaps extend Ada so that some of the detailed design is possible within an essentially Ada context.

Afterthought

Let me close with a philosophical thought. I sometimes think that my counterparts here feel that after an easy year in academia, we fellows should be made to do some honest work during the summer. On the other side of the coin, many fellows will agree with the sentiment found back home in our departments that, after working hard during the academic year, its nice for you fellows to get a paid vacation at NASA. With this in mind, I leave you with this quote.

"It is impossible to enjoy idling thoroughly unless one has plenty of work to do."

J. K. Jerome

Postscript

(Added in press) At several places in this report it has been noted that further coding was stymied by a lack of knowledge of the breadboard for the core module, apparently due to its nonexistence. Surprisingly, and unknown to this fellow during most of the time while he was engaged in this project, there are <u>two</u> working models of the common module in building 4755. The module being built by NASA already has two units to recover carbon dioxide and an oxygen generator.

The core module built by Martin-Marietta is extensive.

In addition to carbon dioxide recovery and oxygen generation, there are also power handling units at three different frequencies, heat removal and automatic balancing, human waste disposal and a trash compactor. There is model software, originally written in C and recently transported to Ada! A user interface is provided.

What has been learned from designing, constructing, programming and operating these two modules should be compared with this report.

If information on these core modules had been provided to this faculty fellow early in the summer, this report would definitely be different.

REFERENCES

[ACD] **Architectural Control Document -- Data Management System**. Space Station Program Office, Johnson Space Center, NASA. Jan. 1987.

[ACGE] Christine N. Ausnit; Norman H. Cohen; John B. Goodenough & R. Sterling Eanes. **Ada in Practice.** Springer-Verlag, 1985.

[Bran] A. E. Brandli. **Operations Management System (OMS)**. Viewgraphs, prepared in the Avionics Systems Division, Johnson Space Center, NASA. Aug. 1986.

[Belk] Boumediene Belkhouche. Compilation of Specification Languages as a Basis for Rapid and Efficient Prototyping. **Third International Workshop on Software Specification and Design**, Aug. 26-27, 1985, London. IEEE Computer Science Press. 1985.

[BGS] Barry W. Boehm; Terence E. Gray & Thomas Seewaldt. Prototyping versus specifying: a multiproject experiment. **IEEE Transactions on Software Engineering, SE-10** (May 1984), pp. 290-302.

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

[Frigo] G. Vittorio Frigo. Evaluation of the VAX Ada compiler and APSE by means of a real program. **Ada Letters 7** (May, June 1987), pp. 94-106.

[GC] R. M. Gallimore & D. Coleman. Algebra in Software Engineering. **Third International Workshop on Software Specification and Design**, Aug. 26-27, 1985, London. IEEE Computer Science Press. 1985.

[HI] S. Hekmatpour & D. C. Ince. Rapid software prototyping. In **Oxford Surveys in Information Technology 3**, P. I. Zorkoczy, ed., Oxford Univ. Press, 1986. pp. 37-76.

[IEEE] IEEE Computer Society. **Ada as a Program Design Language**. IEEE Standard 990-1987. (To appear 1987).

[LNR] David C. Luckham; Randall Neff & David S. Rosenblum. An environment for Ada software development

based on formal specification. **Ada Letters 7** (May, June 1987), pp. 84-93.

[Kno1] Arthur Knoebel. Analysis of high-order languages for use on the Space Station application software, in **Research Reports -- 1985 NASA/ASEE Summer Faculty Fellowship Program**. NASA CR-178709, Jan. 1986.

[Kno2] " " Benchmarks of programming languages for special purposes in the Space Station, in **Research Reports -- 1986 NASA/ASEE Summer Faculty Fellowship Program**. NASA CR-178966, Nov. 1986.

[Kno3] " " A tripartite specification of abstract data types. (Article submitted).

[Kno4] " " The Algebraic Theory of Abstract Data Types. (Book in preparation).

[NW] John Nissen & Peter Wallis. **Portability and Style in Ada**. Cambridge Univ. Press, 1984.

[RFP] Request for Proposal. **Software Requirements Specification, Attachment A for the Space Station United States Laboratory Module**. SS-SPEC-0002, Marshall Space Flight Center, 15 Dec. 1986.

[Rog] M. W. Rogers. **Ada: Language, Compilers and Bibliography**. Cambridge Univ. Press, 1984.