# An Ada Programming Support Environment

Al Tyrrill
A. David Chan

North American Aircraft Operations
Rockwell International
Lakewood, California

ABSTRACT

This paper describes the toolset of an Ada Programming Support Environment (APSE) being developed at North American Aircraft Operations (NAAO) of Rockwell International. The APSE is resident on three different hosts and must support development for the hosts and for embedded targets. Tools and developed software must be freely portable between the hosts.

The toolset includes the usual editors, compilers, linkers, debuggers, configuration managers and documentation tools. Generally, these are being supplied by the host computer vendors. Other tools, for example, pretty printer, cross referencer, compilation order tool and management tools have been obtained from public-domain sources, are implemented in Ada and are being ported to our hosts.

Several tools being implemented in-house are of interest, these include an Ada Design Language processor based on compilable Ada. A Standalone Test Environment Generator facilitates test tool construction and partially automates unit level testing. A Code Auditor/Static Analyser permits Ada programs to be evaluated against measures of quality. An Ada Comment Box Generator partially automates generation of header comment boxes.

## 1 INTRODUCTION

Rockwell International North American Aircraft Operations (NAAO) is constructing a facility for the development of Ada software. The facility will support an avionics integration laboratory where both simulation and embedded avionics software are to be developed. Ada software development will occur on three different hosts.

1. A supermini widely used in the aerospace and scientific communities.

2. Another supermini noted for high "number crunching" horsepower. This processor model will support the simulations and simulation development.

3. A processor designed specifically for Ada software development, on which all system software has been implemented in Ada.

Each of the development hosts will interface to a user maintenance console that supports several of the embedded avionics processors. The maintenance console can pass data between the target processor memories and the hosts and control execution of the targets.

The avionics processors are connected to each other, various actual aircraft hardware and the simulation host by means of several high speed data busses. Software in the avionics processors can be tested with actual hardware online or with hardware simulated by models in the simulation host.

The hosts are to be networked with an Ethernet line so that software, associated products and development tools can be easily transported.

Rockwell is constructing an Ada Programming Support Environment (APSE) for the development facility. The APSE consists of a set of tools whose objective is to support the production of a well-organized, structured and maintainable software product, in a cost effective manner. The APSE itself must be constructed in a cost effective manner.

The cost requirement on the APSE dictates that available tools be used as much as possible. This reduces the potential level of tool integration, as tools implemented in isolation from each other generally will not share common interfaces. The interface that is shared by most of the tools is the Ada language, however, and its rigid standardization makes assembly of a toolset from disparate sources feasible.

## 2 APSE COMPONENTS

This section summarizes the components of the NAAO APSE and indicates the sources from which the tools will be obtained. Section 3.0, Locally Developed APSE Components, describes in more detail some of the components that are to be implemented at NAAO.

### 2.1 Development Tools

These tools support the design and coding phases of the software development process. They are an Ada Design Language, text and program editors, compilers and assemblers, a library of primitives and common packages, and link editors.

#### 2.1.1 Ada Design Language

The objective of the NAAO Ada Design Language (DL) is to provide a means of expression for both control flow and data structure and relationships. The Ada language itself provides an excellent means for expressing data structure, but some other means of describing control flow is necessary prior to actually committing a design to Ada code.

Accordingly, the Ada DL uses compilable Ada to represent data structure and a traditional Program Design Language (PDL) to represent control flow. The PDL statements are embedded as comments within the Ada specifications so that the entire Ada DL description is compilable. Several tools are available to support construction of Ada DL designs. These include a "TBD" package, the Ada DL preprocessor, the processor for the traditional PDL and an Ada body part generator.

The Ada DL is described, along with its use in object oriented design, in more detail in section 3.1, Ada Design Language.

2.1.2 Editing Tools

Several tools support the editing of Ada DL, Ada code and documentation files.

2.1.2.1 Text Editors - Text editors are provided for editing of documentation and other non-Ada files. These were obtained with the system software on each of the hosts.

2.1.2.2 Ada Syntax Sensitive Editors - A syntax sensitive Editor is one that contains the syntax equations of the target language in its database. Templates are expanded to their syntactic substructure. The means exists to traverse between templates and delete templates for optional constructs.

Two of the three hosts have Ada syntax sensitive editors available from the system vendor. In one of these, initial entry of a file begins with the template [compilation], which by repeated expansion and replacement of templates with text, is converted to the desired code. The templates have the same names as the syntax equations from the Ada LRM. When adding to an existing file, it is necessary to enter the starting template e.g. [later_declarative_item], [statement] manually (and one must know what they are called).

On the Ada based host, a construct is prompted by entering an initial keyword, e.g. "procedure", "if", and requesting the editor format the file. It identifies the construct and expands it into its components.

2.1.2.3 Source Formatter (Pretty Printer) - The source formatter reformats existing Ada source into a consistent form. The level of statement indentation is made proportional to the nesting depth. Spaces and line breaks are added to improve readability. Declarations and line comments are aligned where appropriate. The source formatter was obtained from a public domain source, is written in Ada and will be modified to improve functionality. On the Ada based host, the source formatter is integral with the editor.

2.1.3 Compilers and Assemblers

Program development will occur in different environments in the development facility. Native mode code will be generated for initial program testing and for tool implementation. Code will be generated for the simulation host on that host. Ada written for the simulation host must interface with existing FORTRAN code. Ada code will be written for the embedded processors. This code must interface with existing JOVIAL code.

2.1.3.1 Ada Native Mode Compilers - Each of the development hosts has a validated Ada compiler available from the system vendor. Each has an associated library manager for creating and maintaining Ada program libraries.

2.1.3.2 Ada Embedded Processor Cross Compiler - Cross compilers for the embedded processor are available or will be available for all our development hosts, although none have been validated. For two of the hosts, the system vendor will be supplying the cross compiler. for the other, one of several possible third party vendors will be selected.

The different vendors products are currently being evaluated. The selected product will hold a validation certificate or otherwise have been demonstrated to correctly compile those features required by the avionics software.

2.1.3.3 JOVIAL Avionics Processor Cross Compiler - This compiler will translate JOVIAL to the object code of the avionics processor. The object file format will be compatible with that generated by the Ada compilers for the avionics processors. It will be possible for JOVIAL to call Ada and vice versa without the use of interface routines when the parameter types have analogues in both languages. The JOVIAL cross compiler will be obtained from the Ada cross compiler vendor.

2.1.3.4 Avionics Processor Cross Assembler - These assemblers will run on the hosts and generate avionics processor object code. The object file formats will be compatible with that generated by the Ada compilers for the avionics processors. The Ada cross compiler vendors each have compatble cross assemblers available.

2.1.3.5 Simulation Host FORTRAN Compiler - The native mode FORTRAN compiler on the simulation host will generate object files compatible with those of that host's Ada compiler. Such a compiler is available from the system vendor.

2.1.4 Library of Primitives and Common Packages

The library of primitives and common packages will be a collection of commonly used functions in the areas of navigation, weapons delivery and math functions. Initially, a set of primitives will be identified for inclusion in the library and implemented when they are first needed. Additional primitives will be developed as the need for them is identified.

Some type of "browser" utility that will enable the potential user to intelligently search the library is being planned.

2.1.5 Link Editors

The linkers in the APSE shall have the means to determine that all modules dependent on a module that has been recompiled have also been recompiled, or that otherwise the full set of object modules involved in the link edit is in a consistent state.

2.1.5.1 Host Link Editors - These linkers will link object files produced by the hosts' native mode Ada compilers to produce an image executable on the host. Each host system vendor has augmented its link editor to provide the

required consistency checking.

2.1.5.2 Avionics Processor Cross Linker - These linkers will generate executable avionics processor images from object files produced by the Ada cross compiler, the JOVIAL cross compiler and the avionics processor cross assembler. The avionics processor images can be executed interpretively by simulators on the host or downloaded to an avionics processor.

## 2.2 Testing Tools

The Ada environments on the hosts will be integrated with the symbolic debuggers provided with the hosts' operating systems. Symbolic debuggers will be procured for the avionics processors which will support standard debugging operations without incurring additional overhead in the target. A tool will exist to create an environment in which to test Ada compilation units in a standalone mode.

A data bus monitor will support the capture and display of selected bus data and the simulation of bus transmissions to facilitate integration testing.

The development host will have the simulation and support tools necessary to execute the avionics software in an integrated mode with the actual or simulated aircraft hardware or in a software environment only. This includes a host simulator designed to execute flight software in native code supported by environment programs and I/O simulated in software. The hosts will have target processor simulation including input/output and interrupt simulation.

### 2.2.1 Host Symbolic Debuggers

These tools, used for debugging native mode programs on the hosts, supports examination and deposit, setting of breakpoints and watchpoints, stepwise execution and trace, all referenced to Ada source statements or declarations. The debuggers are part of the host system vendors' software support packages, but each has been modified to support tasking and other unique features of the Ada language.

### 2.2.2 Avionics Processor Interface and Debugger

These tools supports downline load of executable images to the avionics processors, execution control of the avionics processors and transmittal of status information back to the hosts. Symbolic debugging is supported from the hosts. Symbol table information is maintained in the hosts and not downloaded to the avionics processors. Target debugger support is provided by all the Ada cross compiler vendors, but additional interfacing to support NAAO's particular test environment will be required.

### 2.2.3 Standalone Test Environment Generator

This tool determines the inputs, outputs and external entry points of a set of Ada programs under test. The tool prompts the user for inputs, executes one of the specified programs and displays the outputs. Pre-canned functions can be specified for the inputs and the program executed repeatedly with variation of an independent variable, such as time. Outputs can be plotted against inputs

or the independent variable.

The Standalone Test Environment Generator will be implemented in-house at NAAO, and is described in more detail in a subsequent section.

### 2.2.4 Data Bus Monitor

The bus monitor will interface with the various data busses in the avionics integration laboratory and perform the following functions. The bus monitor will be implemented by augmenting existing capabilities.

Generate real time displays of selected bus data.
Generate profiles of bus data by message type and subtype.
Generate simulated bus data for test stimulation.

### 2.2.5 Host Avionics Processor Simulator

Simulators for the avionics processors will be available to support the testing of avionics processor images that would otherwise require the actual hardware. A conventional simulator will interpret executable images down to the instruction field level. A faster simulator in which the Ada code is compiled into procedure calls on the host that duplicate the computations of the avionics processors without actually interpreting at the bit level is also being acquired. Both of these are available from the Ada cross compiler vendors.

A simulator is being implemented in Ada inhouse that will be capable of concurrently simulating several avionics processors, with interprocessor communications implemented as transfers through common memory buffers.

### 2.2.6 Documentation Support Tools

The documentation generators will construct data dictionaries from sets of Ada programs. They will construct trees of calls and context references (WITH's). A header comment box generator will summarize that information in the program headers that can be extracted automatically from the program source. These processors will accept a list of files, or scan a link editor command file and process the sources for all the input modules for the linking of the executable image. A report formatter/word processor will be available for general document preparation.

2.2.6.1 Ada Data Dictionary Generator - This tool scans a set of Ada program source files and records the full context of declarations, recognizing Ada scope and visibility rules. It generates a data dictionary with locations of declarations, set references and use references in a format compatible with required documentation. This will be implemented by augmenting public domain software, implemented in Ada.

2.2.6.2 Ada Called-by/Withed-by Generator - This utility does a scan of a set of Ada source programs. For subprograms it constructs trees of calls and called-by references. For packages, it constructs trees of context clause (WITH statement) references. The generated reports are in a format compatible with required documentation. This tool will also be obtained by augmenting an

existing public domain program.

2.2.6.3 Header Comment Box Generator - This tool scans the source of an Ada compilation unit for that information which is required to be in the header comment box, such as inputs and outputs, subprograms called and called-by, imported data structures and routines, and other resources used. It creates a new header comment box or updates the existing one. This tool is being implemented in-house in Ada and is described in more detail in a subsequent section.

2.2.6.4 Report Formatter - These utilities process a file of text with interspersed formatting commands. They perform word processing functions such as right margin alignment, indentation, assignment of heading numbers, table of contents generation and others. Several of these are already installed in our facility, from various sources. They are the most widely used support tools in the laboratory.

2.2.7 Configuration Management Support

These tools support the adherence to software standards and the controlled maintenance of source and documentation files.

2.2.7.1 Code Auditor/Static Analyser - The code auditor scans the source for an Ada compilation unit and generates a report of areas of non-conformance to software standards, as specified in the Ada Style Guide that was developed jointly by several Rockwell divisions.

2.2.7.2 Configuration Control System - The configuration control system will create and maintain libraries of controlled files, which can be Ada DL source, program source, documentation or any other textual material. It will track changes by associating them with retrieval and replacement of library elements. It will monitor access and be able to generate a historical record of the accesses to each element in a library. Each host has such software available from the system vendor.

3 LOCALLY DEVELOPED APSE COMPONENTS

The following sections describe in more detail some of the tools that are being implemented in-house at NAAO. Of particular interest are the following.

1. Ada Design Language processor, that will permit embedding a traditional PDL within compilable Ada specifications.

2. Standalone Test Environment Generator, that will determine the inputs and outputs of a program under test, then generate input values, execute the program and capture and display the outputs.

3. Code Auditor/Static Analyser, which will permit Ada programs to be checked for conformity with software standards, and be evaluated against various measures of quality.

4. Ada Header Comment Box Generator, which will automate collection of some of the information required to be in the header comment box of program units.

## 3.1 Ada Design Language

Traditional PDLs, like those widely used in the computing community over the past decade are good at describing control flow, but poor at describing structure, hierarchy, data relationships and interfaces.

Ada specifications are good at describing these things, but do not describe control flow. Use of compilable Ada to describe control flow is awkward, at best, because it does not permit specification of detail to be deferred.

The idea of using compilable Ada as a design language is gaining acceptance because it specifies at design time what the software product will look like. I.e. the Ada specs are a form of "contract" for the software that is to be implemented.

Traditional PDLs are coming to be regarded as a decade old technology that is little more than an improvement on flowcharts.

The NAAO Ada Design Language combines compilable Ada with Reconfigurable Design Language (RDL), a traditional PDL with an Ada-like syntax, to obtain the benefits of each. RDL was implemented at another Rockwell division in Ada and can be installed on any host with a validated Ada compiler. Aside from the syntax change to make it more Ada-like, it is similar in appearance and capabilities to a commercially available PDL widely used in the computing community for over a decade.

### 3.1.1 Use of the Ada DL

Use of this design language consists of the following steps.

1. Description of the structure, operation and interfaces of a design using Ada specifications.

2. Construction of the Ada bodies, starting with the specifications, then with further development.

3. Description of the control flow within units, using RDL statements in specially marked comments.

3.1.1.1 Development of Ada Specifications - The design language user first identifies the objects to be implemented. These suggest the top level package structure of the design. Then, the actions to be performed on these objects are identified, these suggest the procedures and functions these packages will support.

Externally visible data structures are identified, then Ada types and objects are defined to represent these. Parallel event streams suggest creation of tasks to support them. Textual comments are added to further explain the purpose of the constructs so defined.

3.1.1.2 Development of Ada Bodies - Then, using an Ada body part builder, body parts for the specifications are created. Data structures not to be visible externally are defined within the bodies of the packages and subprograms. Use of the available "TBD" package permits the user to defer assigning specific types to Ada objects.

3.1.1.3 Development of RDL Descriptions - The control flow within the subprogram bodies is now designed and specified with RDL procedures. The RDL statements are enclosed in specially marked Ada comments to keep the entire design description compilable. Reference to data defined in the pure Ada part can be made by the RDL. Use of RDL permits the existing RDL processor to be used to generate data dictionaries and calling trees.

Large designs may require several iterations of this process before the design is complete. The completed design consists of Ada specs with embedded textual comments and Ada bodies with embedded RDL procedures and comments.

3.1.2 Ada Design Language Utilities

Several tools and utilities are available to assist in the generation of Ada DL descriptions.

3.1.2.1 TBD Package - This TBD package, which is public domain software, provides types, objects, functions and a procedure which can be referenced in a design when the actual type of the object or subprogram parameter is not known. TBD values for the quantities in package SYSTEM, such as maximum integer, smallest fixed point delta, etc. are also defined. As a design is evolved, the TBD quantities are replaced with the actual objects. All names in the package contain the substring "TBD" so they can be located with an editor search.

3.1.2.2 Body Part Generator - This tool generates a body part from an Ada specification. It is available as a primitive on the Ada based development host, and also from a public domain source for any processor with a validated Ada compiler.

3.1.2.3 Ada DL Preprocessor - This utility, which will be implemented in-house, scans the Ada Design Language descriptions and records all the type, object, subprogram and task specifications. It extracts the RDL procedures from the Ada bodies and generates RDL declarations for the objects declared in the Ada and referenced in the RDL. It formats the RDL into a form acceptable to the RDL processor and submits it for generation of an RDL report.

3.1.2.4 RDL Processor - The RDL processor, currently installed on two of our hosts, generates a formatted report from an RDL description. It also produces a data dictionary and calling trees for the segments (subprograms).

3.2 Standalone Test Environment Generator

Traditionally, unit level testing is done by implementing special purpose data generators and data monitors, linking everything together, running the program under test, then analysing the data. The next routine requires new data generators and monitors.

The Standalone Test Environment Generator (STEG) being developed at NAAO will act as data generator and monitor for a large class of subprograms and will partially automate the unit test process.

A unit to be tested includes a subprogram and its dependent units. They are first compiled cleanly.

The STEG will scan the unit under test, identify the calling parameters, then determine those objects declared at a higher scope that are used but not set (inputs) and set but not used (outputs). It will detect those that are both set and used, as these could be inputs, outputs, both or neither.

The STEG will prompt the tester for the names of inputs and outputs not identified in the scan. It will then generate an Ada shell that supplies the program's inputs and captures its outputs. This will be compiled and linked with the program under test. Stubs will be provided for subprogram that are not provided. An OUT parameter from a stub is regarded as an input.

The STEG will then prompt for the values of the identified inputs and pass them to the target program. It will execute the program under test, then display the values for the identified outputs. Exceptions returned from the target program will be identified. Facility to generate an exception from a stub will also be supported.

A command language will be provided for repeatedly executing the program under test while varying the values for selected inputs. The command language will be a subset of Ada. Plotting and data reduction features are to be provided.

### 3.3 Code Auditor/Static Analyser

The purpose of this tool is to support the enforcement of software standards and good programming practices. It will gather statistics that may be indicative of the use or non-use of these standards and practices and prepare a report that might serve as the starting point for a code review or structured walkthrough. The code auditor will gather the following types of statistics.

1. The amount of commentary relative to the amount of code will be determined. Textual comments will be distinguished from delimiting comments (blank lines and lines of dashes, etc.). Of course, it will be unable to distinguish a useful comment from something like "-- Mary had a little lamb".

2. Measures of program complexity will be developed, such as number of nodes in a program's directed graph, then statistics will be developed from our experience with implementing and maintaining these programs relative to their measured complexity.

   It is generally regarded that overly complex program units cause maintenance problems. However, simpler programs mean more program units are necessary, and this complicates the integration process.

The number of subtypes and derived types defined and their frequency of reference versus the frequency of reference of the predefined types. Use of subtypes and derived types makes better use of the strong typing features of Ada.

4. For subprograms, the number of parameters passed versus the direct references to objects declared at a higher scope (global variables). Use of global references is regarded by some as producing harder to read code.

5. Statistics on identifier length will be gathered. Average length, distribution of lengths and frequency of reference of various lengths will be recorded. These statistics will be gathered for various classes of identifiers, e.g. scalars, record components, FOR loop indeces, subprogram formal parameters, etc. Use of overly short identifier names is regarded as a poor practice, but it is not clear that longer is always better.

6. Use of PRAGMAs, particulary PRAGMA SUPPRESS, will be recorded and summarized.

7. Placement of more than one type or object declaration on a line, or more than one executable statement on a line will be flagged. Code so written is likely to be harder to read.

8. Types and objects declared but not referenced, objects declared at a higher scope than necessary and uninitialized objects will be flagged.

9. The number of declarations and executable statements for each subprogram will be recorded. These values will be provided both including and excluding nested subprograms. The maximum nesting depth for control structures, subprograms and tasks will also be determined for each program unit.

10. The number of GOTOs and jump target labels (<<LABEL>>, not LABEL:) will be counted, and a measure of the "branching complexity" of a routine will be determined.

11. Unlabelled blocks and loops will be flagged. Use of these labels often provides a valuable form of commentary.

12. Declaration of typed constants vs. universal numbers will be flagged when appropriate. Use of a DELTA other than a power of 2 for fixed point types will be detected. Use of a radix other than 2, 8, 10 or 16 will also be flagged.

### 3.4 Header Comment Box Generator

Software standards at NAAO specify that each compilation unit be headed by a comment box that contains detailed information about the unit.

Among other things it is required that the comment box list all sets and references to global variables (objects declared at a higher scope), all subprograms and tasks called, task entries, exceptions generated (other than the usual Ada exceptions) and exceptions handled, and all packages, tasks and

subprograms defined internally.

The header comment box generator will detect the presence of these features and create the part of the header comment box that describes them. If already present, the existing comment box will be revised.

Gathering this information for the comment box is a tedious task which implementers do without enthusiasm, and thus without attention to correctness and detail. Frequently the information is ignored when it is needed (say, by a tiger team called in to fix a high-priority problem) because it is assumed to be incorrect and out of date. Automating its generation should greatly increase its reliability and usefulness.

## 4 CONCLUSIONS

The Ada development environment described here meets most of the needs of our current and near future development requirements. The objectives of a cost effective APSE implementation and a versatile development environment are expected to be satisfied.