ADA AND LARGE REAL-TIME SYSTEMS IN THE ROYAL NAVY

BY

D. W. DAVIES, B.Sc.(ENG.)
(Admiralty Research Establishment, Portsdown)

ABSTRACT

The Ada programming language was developed in the 1970s and introduced in the 1980s. This article explains some of the processes necessary for its introduction and the problems resulting, and describes work being done by ARE to measure these problems and propose solutions.

Introduction

During the past thirty years many computer programming languages have been used for the implementation of real-time naval systems. In my own field—Naval Command and Control Systems and Weapon Systems—this variety has included machine-code, assembly code, Ferranti autocode, CORAL 66 and now Ada. This is only a small subset of the usage throughout MOD. This variety of languages poses great problems for both development and maintenance. The requirements to reduce these problems are the driving force behind the development of the Ada programming language for real-time computer systems by the U.K. MOD and the U.S. Department of Defense (DOD).

Ada Programming Language—history

The Ada programming language was commissioned by the DOD after several years study and development work. The language itself was designed by CH Honeywell Bull after a competitive proposal to the DOD.

The study started with a document called STRAWMAN published in 1975 which outlined the requirements. After discussion and comments this was further refined and was called WOODENMAN, which was similarly refined to produce TINMAN in mid-1976. Several current languages were examined against the TINMAN requirements to see if any could be developed to meet the requirements. Nothing suitable was found. Further refinements were made to the document and IRONMAN was produced. Proposals to design a new language were invited from industry. Seventeen proposals were received and four were chosen to go ahead in competition. Initial designs were received in early 1978 and two of these were judged to have sufficient merit to warrant further development. At this point further refinements to the requirements document were made to produce STEELMAN. After another year of development the final choice was made on 2 May 1979. After a further year of presentation and discussion the first definitive version of the language was published in July 1980, and it was proposed to the American National Standards Institute (ANSI) as a standard. The standardization process took two years and resulted in a few changes to the language. The ANSI standard Language Reference Manual[1] was finally published in January 1983. The rules of ANSI say that a revision exercise should be undertaken and a revised standard document produced after ten years. This revision exercise has now started as an international exercise. It is known as Ada 9X and expects to

produce a revised standard in 1993. A more detailed history can be found in chapter 3 of Booch's book[2].

Ada is based on PASCAL and is the first practical language to bring together important features such as data abstraction, multi-tasking, exception handling, encapsulation and generics. It was originally intended for 'embedded' systems but it is equally suitable for use as a general purpose language.

MOD Computing Policy

The MOD Computing Policy document was first published in December 1987 [3] and is the result of discussions between MOD and Industry. It is intended for the information and guidance of Project Managers within the Procurement Executive and for companies bidding for contracts relating to computer-based systems. This policy document states that MOD preference is that software intensive projects should use the programming language Ada for implementing the system. The original policy is being revised in detail.[3] The policy covering the programming language has not changed. Ada is the preferred language, the compiler used must be a validated compiler and must also have been examined using the MOD-sponsored evaluation process, AES, and the report must be available to the MOD Project Office.

NATO has also adopted Ada as a preferred programming standard. The MOD has proposed to NATO how this policy could be implemented. This proposal includes the following three items, 'Waiver from Policy', 'APSE' and 'Minimum APSE Toolset Configuration'. In this context APSE is short for 'Ada Project Support Environment' and the last item covers allowable APSE configurations depending on the size of software system to be developed. The NATO policy document is still under discussion within the NATO Information Systems Working Group.

Validation

Validation is a process of measuring a compiler and to a very, very small extent the associated runtime system to ensure that this all conforms to a recognized standard (DOD STD 1815a). As custodians of this language the DOD has commissioned a suite of test programs called the Ada Compiler Validation Capability (ACVC). This validation suite is a collection of test programs written in Ada that is applied to the vendor's Ada compiler suite. The validation suite is composed of several thousand tests (version 1.9 contains 3122 tests) and this number is still increasing. It needs to be emphasized that the Validation test suite does not attempt to examine the performance of the Ada runtime environment. The compiled code is run, of course, to ensure that runable code is produced. The validation process is supervised by the Ada Joint Program Office (AJPO) which is a U.S. DOD office. The certificate that is issued after successful completion of the tests lasts until the next version of the validation suite is in service and confirms that this issue of the compiler has passed the current version of the validation suite. The version of the validation suite is changed once every two years (version 1.9 expired on 1 June 1988), and is available to compiler vendors about six months before its official start date. The certificate is issued to compiler vendors upon satisfactory completion of the test suite and payment of the validation fee which is currently about £10 000.

The U.K. has obtained agreement with the DOD that the National Computing Centre (NCC) is able to carry out the Validation process and to recommend to MOD the issue of Validation Certificates. The MOD is the only place outside the United States that is able to issue these certificates. There are however several other countries able to carry out validation tests, such as France and West Germany.

The MOD has proposed to NATO a policy to cover Ada Compiler Maintenance to take care of the fact that Ada compilers will need to be revalidated every few (two at present) years and this may imply significant changes to the compiler during the lifetime of the software system, both development period and in-service use. This proposal was issued by Eng Pol 31 RAF under reference D/D Eng Pol(RAF) 57/5/12/1; and advice is given to Project Managers on the selection of an Ada compiler in 'The Use of Ada Compilers in MOD(UK) Projects' dated 5 October'87 reference D/D Eng Pol(RAF) 100/31/1.

Evaluation

This process is aimed at examining an Ada compiler and its runtime system to measure how suitable a particular compiler is for a general purpose Application. There is a U.K. MOD-sponsored Evaluation process (AES). The U.S. DOD has also commissioned an Ada Compiler Evaluation program (ACEC). There are also several other sets of code designed to examine in detail features of high-level language compilers, e.g. accuracy of floating point arithmetic and speed of matrix manipulation. In theory all these test programs could be used to gain insight into the quality of a compiler; in practice however so much data will be produced from the tests that it would take too long to read and come to a decision.

U.K. AES

This is a special program designed to examine several parts of an Ada compiler and its associated runtime environment (RTE). The tests are mainly aimed at the runtime performance of the compiled code and there are about 200 tests. The tests are arranged in groups, each group devoted to a particular facet, e.g. compilation speed, arithmetic accuracy, capacity of compiler and some speed measurements. The first issue of this test suite appeared mid 1987 and the system is being further developed by Software Sciences, Ltd. British Standards Institution (BSI) handles the commercial aspects of the test in U.K.

The test suite is available from BSI at a cost of £1500 plus VAT; this is fully refundable against the first full evaluation fee. The suite will be maintained free of charge for the first year; subsequent years maintenance cost £300 per year.

For evaluations carried out by BSI they will provide a copy of the test suite to the client so this can be configured for the host computer, compiler and target to be examined, and will later make a formal examination of the results from the tests. A formal report will be published that will be available to intending users of the tested facility. The current fee for this evaluation is about £12 000. The formal report is a bulky document containing the results of all the individual test programs and, in order to make this information more digestible, BSI will prepare a short managerial summary.

The MOD evaulation suite is available now for commercial evaluation; five evaluations have been carried out and the reports are on sale from BSI at £250 each.

DOD ACEC

The U.S. is also preparing an evaluation suite. It is aimed at performing a series of 'bench-mark' tests on the compiling and runtime facilities. The major contractor on this evaluation suite is Boeing Military Airplane Company and the first version was released in September 1988 with a second version due in September 1989. No information is yet available what evaluation tests have been carried out and with what result. The American view is that this should be available as a test suite for all those interested

instead of developing a government evaluation agency. After evaluation and analysis of the results is complete the product evaluated will be awarded a score. It is intended that this score/figure will be used by contractors and project offices to help choose the compiler/host/target configuration that will best fit the project in hand. Only time will tell whether this attitude produces a better result than the MOD one.

The ACEC consists of approximately 240 test progams comprising over 100 tests and some support tools to analyze the test results. The main emphasis of the ACEC is execution performance, but it also addresses compile-time efficiency and code size efficiency. ARE is trying to obtain a copy of the ACEC with a view to compiling a list of the tests available and so reduce the amount of work required to develop a comprehensive run time evaluation suite.

ADA LANGUAGE TECHNICAL ISSUES

The following sections are devoted to the features of the programming language and its supporting tools. The purpose is to expose some of the problems associated with the use of the language in naval projects and so warn prospective users and development managers that there are pitfalls to be avoided.

Reliability Matters

The Ada language is described in the language reference manual[1]. The following features are all taken as being good features which significantly enhance the language. These three features may not be the only good features and others will be referred to by the Ada community in order to emphasize one point or another. The question of software reliability is a controversial matter, depending not only on development techniques but also on design decisions. The use of reliability in this section is more to do with robustness and error-free code that implements very closely the design.

Declaration of Variables

As with other programming languages whenever a variable is declared a data type can be assigned to it with Ada. Also in Ada it is possible, in fact overtly encouraged, to invent new data types and the range of values associated with this new type. The compiler will ensure at compile time that all statements and expressions conform to the rules associated with data types and produce error messages when these rules are broken; it will also insert code to check at run-time that only values within the defined range are used with the variables. Any departures from these rules detected at run-time will show up as 'Exceptions' (see below). The unvarying adherence to these typing rules gives this language advantages over other languages where the compiler is allowed to guess what the programmer really meant and to do type conversion if necessary.

Information Hiding

The language embraces some of the concepts of modular programming by the use of 'packages'. It has to be emphasized that this is modularity at compile time and not necessarily modularity at runtime. A package is a set of declarations and code that are related to a part of the final software system. Each package has a name that is used for calling purposes. The package can be separately compiled and tested during development. Any piece of program that makes use of a package can call for the inclusion of

the package during compilation but only needs to call for those packages that are really required. In this way the contents of packages not called for are not available at compile time and hence this information is hidden.

Exception Handling

During the running of any code it is possible for the underlying/supporting hardware and software to detect errors. These errors may be like attempting to divide by zero, an arithmetic operation has overflowed (addition result greater than 2 to the power 32), or attempting to read/write outside the boundaries of an array. This type of error is described in Ada as an Exception, and Ada code inserts can be written that will handle the error and allow the program to recover from the effects of the error. It is also possible within the Ada source code for the program to raise exceptions when error conditions are discovered by the program. The handling of all these types of Exception is fully controlled by the language, including how the program will behave if the Exception is not handled. Throughout this document when statements are made concerning runtime checks that can be left in or taken out, the checks under discussion are some of the ways in which errors are detected and Exceptions raised.

Safety/Security Critical Systems

The Ada language has no special features to improve the predictability/reliability of the final code over the features described here and the whole reason for the language, which is to improve programmer productivity and target reliability. Any measures taken to further improve quality in this area will have to be taken by the designer/programmer of the system. The MOD is taking steps to change this by preparing two defence standards, and attempting to propose a Project Support Environment standard through the research programme ST(D)2004 R.

Comments

These three subjects (declaration of variables, information hiding, and Exception Handling) are some of the many good features of the language. Strong variable typing rules and information hiding are both measures to improve the robustness of the compiled code. During the development of a program much time is spent debugging the object code. Experience has shown that a lot of debugging time is saved, the majority of programming errors being detected during compile time mainly due to these two features.

The use of Exceptions and Exception Handlers has been discouraged in the past mainly because of the time taken to check for errors in the runtime software and then to change the normal flow of control to the error handling sequences. It is possible to retain these runtime checks in the final operation system or indeed to discard these checks usually in the quest for performance.

In my opinion it is not possible to eliminate Exceptions and Exception Handling completely from the final system because the hardware checks will always be present. I believe there is a great opportunity here for the programmer to make the code more robust; but significant effort needs to be put into the design of the exception system that detects and handles the runtime errors. Of course more time will be spent doing any particular sequence but there are several programming devices in the language that can be employed to reduce the amount of 'exception checking' code the compiler needs to insert; also with the advent of powerful microprocessors there should be sufficient power to devote to these features. A major problem may be that this facility has not been available before and the designers have not had the training to enable them to recognize possible fault conditions or consider how to handle them.

Two draft standards covering the 'safety/security critical' aspects of software generation (00-55 and 00-56) have been circulated for comment before publication. A considerable amount of adverse comment was received and the drafts were subsequently withdrawn for redesign. Nevertheless the general idea that a hazard analysis is done covering the proposed system and special treatment is given to some areas of the system seems sensible. In this context software should not be taken in isolation; the system must encompass both hardware and software. However the techniques for analysis and specification of code for an interrupt driven system containing several threads of concurrent processing are extremely time-consuming and very costly to the extent that large Command and Control systems are impossible to handle at present. Work is in hand in this area but a practical foreseeable end date is not yet in sight.

Real-time Systems

The Ada language is aimed at real-time systems and the compiler vendor is required to provide not only a language compiler but library routines to provide input/output and a runtime environment (RTE) to support concurrent process scheduling.

Parallel Processing

This concept is supplied by the Ada Task structure and the Task scheduling mechanism. The Language Reference Manual (LRM) [1] says Tasks are entities that may operate in parallel with other Tasks. Of course in the simple case of a single processor system only one piece of code can be running at any one time and it is the purpose of the runtime environment to allocate the processor resources to each Task in turn and thus emulate parallel activity. The operation of multiple processor systems is discussed later. The ability for separate Tasks to communicate together is provided by the Ada Rendezvous mechanism. To use this mechanism an Entry is declared in one Task that is Called by another Task. There are the usual array of syntax rules to cover the use of this mechanism. Sufficient to say that when one Task reaches Entry Call and the other Task reaches the Call accept point these two Tasks have performed a 'rendezvous'. At this point data can be passed from one Task to the other in either direction. It is also acceptable for several Tasks to Call an Entry in one Task, and to cover the possibility that several calls are in place at some time a first in/first out queue is associated with each entry. The RTE is responsible for controlling the processor sequencing to achieve this 'rendezvous' effect.

The previous paragraph is a simple description of the Ada 'rendezvous' mechanism. There are many other combinations of circumstances that the language caters for and only a careful study of the LRM will reveal all the possibilities.

Memory Management

It is possible during the operation of any program to share the memory of the processor between two or more items if it can be guaranteed that at any time only one of these items needs to be in play. This feature is already used in the concept of a push down/pop up stack of data. In Ada however it is possible to have not only data that appears and then disappears during execution of the program but also Tasks (i.e. program and data) that appear, start, stop and disappear. As a consequence of this activity it is conceivable that pieces of memory become available for re-use, and also the whole memory becomes fragmented with some sections in use and others not in use. The management of the memory to retain the availability of the whole memory throughout the operation of the program is a complex and time-consuming task.

Priority Systems

In a program containing many concurrent Tasks it is quite likely that the operation of some is more important than of others. To enable the program designer to inform the RTE which Tasks are more urgent than others it is possible to assign a priority value to each Task. The LRM describes how priority shall be prescribed but it is an optional feature. The compiler vendor does not have to offer priority levels neither is he limited as to how many levels he provides.

Scheduling Mechanisms

The scheduling mechanisms can be either 'pre-emptive' or 'co-operative', and the technique of 'time-slicing' may be applied with either mechanism. In time-slicing the RTE divides the usable time into short periods and at the end of each period interrupts the normal program flow and examines whether a reschedule of Ada tasks is necessary and if so activates this change. The program designer is sometimes allowed to specify the duration of the time-slice periods. There is a trade-off in this choice, that is quick reaction to changing circumstances against spending more time checking for rescheduling and context switching. Theoretically in pre-emptive scheduling whenever a task is made eligible to run the RTE will check to see whether this 'new' task has a higher priority than the current task and if so make the necessary context change to run the 'new' task. Occasions for changing the status of a dormant task include, after an interrupt, the expiry of a delay period or the completion of a rendezvous. In 'co-operative' scheduling, once a task has started to run then only when it reaches a scheduling point will the RTE be able to reschedule the resources of the processor.

The scheduling mechanism to be used within an Ada-based system is not prescribed and it is up to the compiler vendor to choose how the RTE shall perform. The only proviso in the LRM is that when a task of higher priority than the current task becomes eligible for running then it is right that this other task should be running. I think this implies 'pre-emptive' scheduling should be used, but this is not how some compiler implementers have interpreted the LRM.

Comments

One characteristic of the Ada Task/Rendezvous mechanism is that it provides a synchronous data transfer process with no buffer storage space. This implies that the data must be consumed at the same rate that it is produced. This may be a suitable mechanism for some data processing systems; but when, say, radar data processing is being done this mechanism cannot be satisfactory. One might say that radar targets are detected at a constant rate of 100 per revolution but there is no way these targets can be forced to be evenly distributed around the radar set; they are much more likely to be bunched up in one part of the sky and thinly distributed in the other. Chapter 14.4 of Barnes's book[4] describes how a buffer storage mechanism can be developed in Ada; but upon analysis this proposal requires the program to be context switched up to four times just to pass a single packet of data from one task to another—this must be very wasteful of resources. The asynchronous data passing mechanism of the MASCOT channel mechanism is a more economic way of providing this inter-Task data transfer facility.

Once again taking the air traffic data processing field of work, a very elegant process is described in Barnes's[4] chapter 14.5. This suggests a single Ada Task is declared that carries out the tracking function for all air tracks, a new instance of this Task is created when a new track enters the playing area, and this Task is deleted when the track leaves the playing area. When

new data associated with this track arrives this Task is activated and eventually scheduled and run under control of the runtime environment. The Task then becomes dormant again waiting for the next piece of data. When the track disappears this instance of the task can be deleted and the memory used can be recovered and allocated to another activity. However the Ada LRM[1] says that the runtime environment should keep some knowledge of the Ada Tasks that have been run and are no longer in use. The amount of space used for this knowledge may be small but over the period of a naval mission of several days a considerable amount of storage will be used up just holding information that is only of use to the runtime system and not to the air traffic control system.

Whenever memory management systems are employed in current online systems, e.g. general purpose multi-user operating systems, the designer of the system has a choice as to whether to attempt to sweep up the data space that is available for re-use whenever it is released or to wait until a suitable point and then collect all the space that has been released. The problem here is that this strategy is wholly within the control of the RTE and the system designer has no control over when it is done. Even if it were possible for the system designer to specify when to do this recovery activity the designer will not be able to say that there will be time now to do the job because he is not present at runtime to assess the actual situation. Similarly the operations team, although they are present at runtime may not be always be able to allow the system time to do garbage collection when the system needs to do it. Quite rightly this is a job that should be under the control of the designer with, maybe some override possible to the operator—it would not be right for the system to 'die' for a second or two during a missile engagement. This is a difficult subject that needs a lot more thought before a suitable answer can be given.

Some compiler vendors may think they are doing the system designer a favour by giving him a large number of priority levels. This, in my view, is not true; a designer with a vast number of priority levels will have great difficulty sorting out the assignment of priority to the various Ada Tasks within the system. There will be temptation to attempt to use many more levels of priority than are really required to make the system work. I suggest that no more than five different levels will be needed in a Command and Control system.

'Time-slicing' is an easy way of providing a scheduling process; everything seems to be under the control of the RTE designer. However this mechanism still causes a delay before a high priority item is run in preference to a low priority item. The maximum delay period is equal to the time-slicing period. The only way to actually get the higher priority item to run as soon as it is eligible to run is by use of the pre-emptive scheduling mechanism; but this mechanism must be properly designed so that it is a true pre-emptive mechanism and there are no holes through which rescheduling instances are missed. The 'co-operative' mechanism is not to be recommended because it is too easy for a programming mistake to provide a low priority Ada Task that could 'hog' the processor by getting into an endless loop which contains no rescheduling point. Special effort has to be made by the system designer to ensure that every Ada Task contains rescheduling points. This imposes extra constraints on the system designer who now has to recognize when long time sequences in the applications code have to be broken into so that other tasks can have a share of the central processing unit (cpu). This requires artificial breaks to be inserted into the target code, i.e. the program has to cooperate with the RTE. There is also no statement in the language that says 'this program can be interrupted at this point if necessary'.