N89-16346

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.

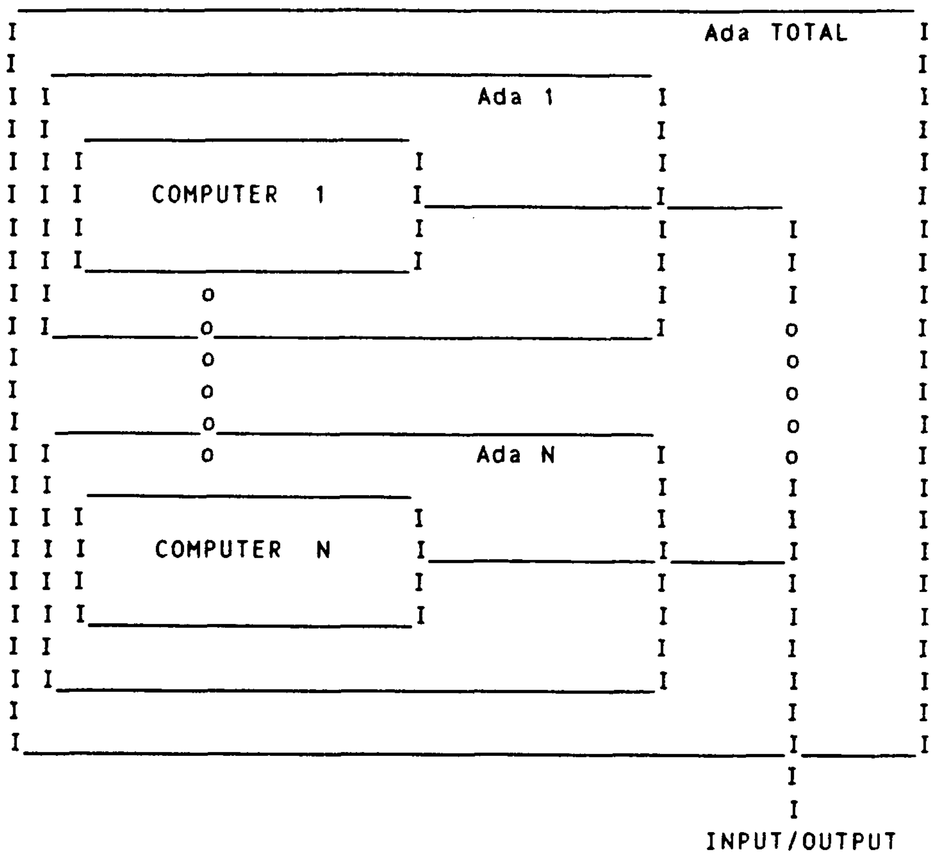
Ada is designed to give low maintenance costs and high quality of the program. Therefore, because the customer pays the life cycle cost, Ada will be attractive for customers with long lived projects, which contain vital functions. However, development of Ada programs requires at least the same or more 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.

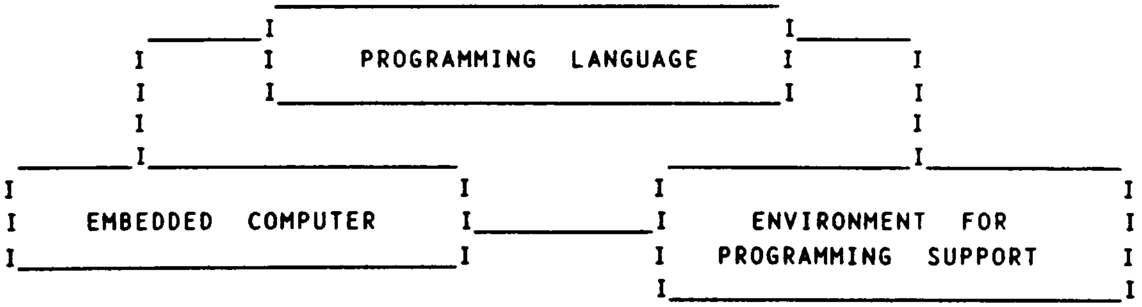
From the computer customer's point of view it seems attractive to choose the Ada TOTAL alternative and let the Ada compiler do a lot of the work. In the Space Station, for example, it would be nice if one "clever" Ada compiler handled the communication between computer modules. However, in practice there 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.

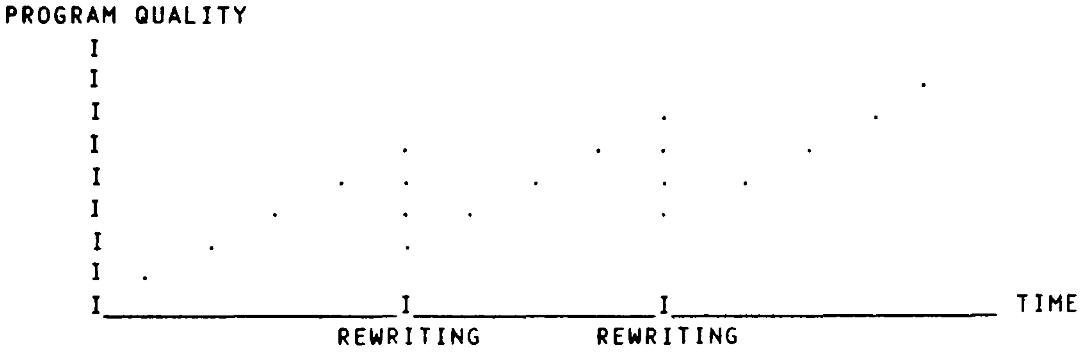


The programming language in fact belongs to the environment, but because of its great importance, it is often handled as a separate component.

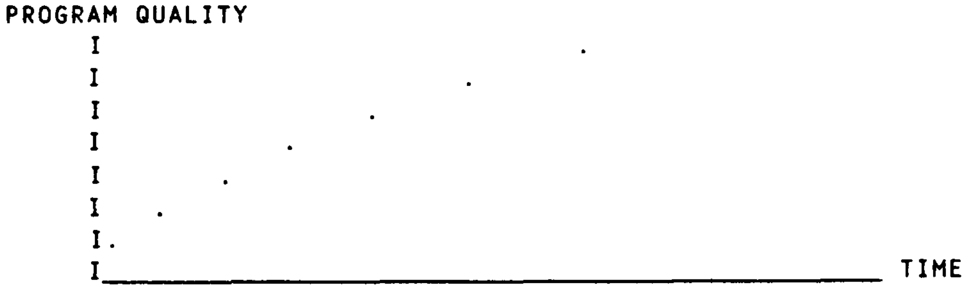
It is important to choose and adapt these three computer system components to each other in a proper way to make it possible to reach specified requirements, for example quality and low life cycle cost.

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.



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:

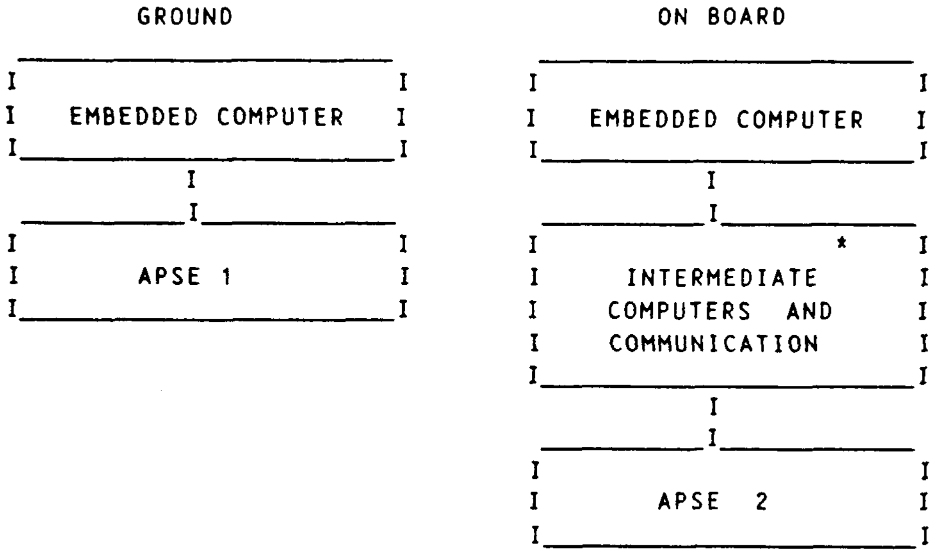


It could be a risk that such standardization and compatible ideas destroy the possibilities for computer hardware evolution, but I do not think there is any risk in practice. An example of this is the MIL-STD-1750A. There are very powerful and usable such processors, even if their instruction architecture 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).



* 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?