AN APPROACH TO THE FORTRAN-TO-ADA CODE CONVERSION PROBLEM

Capt Jeanne Murtagh
Lt Milton Diaz
Mr Roger Turner

AFWAL/AAST
WPAFB OH 45433
253-3215 -4925

ABSTRACT

The Applications Branch of the Avionics Laboratory has a significant amount of modeling and simulation FORTRAN code. We have examined the advantages and disadvantages of converting this FORTRAN code into Ada. This paper addresses our reasons for deciding to convert much-but not all-of this code into Ada. The code conversion methods we considered are discussed, and the method we chose is described.

INTRODUCTION

The Applications Branch of the Mission Avionics Division, Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB OH, performs a significant amount of modeling and simulation work. We have large libraries of FORTRAN code, which we have produced in-house, in addition to models we have purchased. This software runs on a variety of computers, including a DEC VAX 11/780 and a number of microcomputers.

We have examined the advantages and disadvantages of converting these FORTRAN programs into Ada. This paper addresses our reasons for deciding to convert much-but not all-of this code into Ada. It also discusses and evaluates the code conversion methods we considered. The method we chose is described, and conclusions we reached during this effort are presented.

THE ISSUE: SHOULD FORTRAN CODE BE CONVERTED INTO ADA?

There are advantages and disadvantages to be considered if code is to be converted from FORTRAN into Ada.

The most obvious reason to convert code into the Ada programming language is that the DoD would like to stop developing and maintaining code in languages other than Ada. We have been pushing in this direction since the mid-1970s, when Ada was developed in response to the astounding rate of increase in software costs.

It will also be easier for our organization to maintain code in the Ada language than to maintain it in FORTRAN. We have several software models which run on the VAX, and on several microcomputers. Unfortunately, each of these machines has its own dialect of FORTRAN. The VAX supports the full ANSI standard for FORTRAN-77, and then adds a number of Pascal-like features. The microcomputers tend to support only a subset of the ANSI standard for FORTRAN. Programmers maintaining code which must run on several different machines have to ensure that the modifications or enhancements they develop will work in each machine's dialect of FORTRAN. This problem is eliminated when we convert the code to Ada, since an Ada compiler cannot be validated unless it supports all the required features of MIL-STD-1815A.

It is also better for our customer organizations to receive Ada code. Many of the organizations to whom we deliver code do not normally work with FORTRAN. If we convert the code to Ada before we deliver it to them, then these customer organizations do not have to develop a proficiency in FORTRAN before they can understand and use our code. They also do not require FORTRAN compilers in order to incorporate some of our code into their software developments. Since DoD business is moving toward the Ada programming language exclusively, our customer organizations usually already have staff who know Ada. Organizations which do not have Ada knowledge in-house will have to acquire it in order to stay competitive, rather than just to understand our code. Converting the code to Ada before distributing it can reduce the number of programming languages other organizations need to learn, and the number of compilers these organizations require. This can result in significant cost savings.

We considered several problems with converting FORTRAN code into Ada. It will take time and effort to convert models which are already functioning into another programming language, regardless of the method chosen to accomplish this code conversion. We also
considered the potential problems with the quality of the converted Ada code. We decided, during our study of a number of code conversion approaches, that these problems could be minimized or eliminated by selecting a sound technical approach.

We determined that the benefits of having Ada code, instead of FORTRAN code, were worth the investment required to accomplish the conversion for most of our code. We elected not to convert code which contained large numbers of constructs which could not be translated smoothly. We also decided not to convert any code which was maintained by other organizations.

POSSIBLE APPROACHES TO CODE CONVERSION

We evaluated three approaches to the code conversion problem:

1) Manually convert the code from FORTRAN into Ada.
2) Purchase a translator, or translation services.
3) Construct a simple translator.

The advantages and disadvantages of each method are discussed below.

The most appealing feature of the first method is that it promises the best final product. Since this method requires a complete re-write of the code, it provides the best opportunity to replace any unstructured features of FORTRAN with high quality structured Ada. Rather than translating FORTRAN GOTO statements into Ada GOTO statements, the software engineers could evaluate the construct being implemented and replace it with its structured Ada equivalent. If the construct did not match a structured Ada construct, the code could be redesigned during the code conversion process.

The problem with this approach is the investment it would require. The code conversion process would require many man-months. We would also have to extensively retest the software, since it would be essentially completely rewritten. This would also require a significant manpower investment. We determined that we could not afford to use this method, since we have large libraries of code we would like to convert.

We considered purchasing either a translator, or code conversion services. This method is appealing because it would require very little manpower from our organization. However, this option could become very expensive, especially if we contracted for translation services, rather than purchasing a translator. This method would also give us very little control over how FORTRAN features which do not translate smoothly into the Ada language would be handled. Consequently, we rejected this option.

We determined that we could build a simple translator. In order to keep the translator design from becoming too complicated, we decided to automatically translate only those FORTRAN features which can be translated easily into Ada. FORTRAN features which would be difficult to translate automatically into reasonably high quality Ada code would be flagged in the output, and would have to be translated manually.

This, in effect, would be a compromise between the manual and the automatic code conversion options. This simple translator could be produced relatively quickly, and we would retain precise control over how FORTRAN constructs were translated. The construction of this translator would also be a valuable training experience.

This method will not produce Ada code of the same quality as that obtained through rewriting the FORTRAN code. However, we determined that we could produce Ada of acceptable quality. This method would also allow us to save some effort during the regression testing of the newly produced Ada code. Once we had completed thorough testing of the translator, and were confident of its capabilities, we would be able to concentrate our regression testing efforts on code which had required manual translation. We would only need to perform some regression testing on the Ada code produced by the translator, rather than having to test as extensively as if the code were newly designed.

We selected the construction of a simple translator as our approach to the FORTRAN to Ada code conversion problem.

THE SIMPLE TRANSLATOR

We decided to limit the FORTRAN constructs processed automatically by the translator, in order to simplify its design, and to improve the quality of the code it produced.

The translator recognizes all FORTRAN keywords. It can process both logical and arithmetic IF statements. The FORTRAN DO statement is converted into the Ada DO statement. GOTO statements are translated directly into ADA GOTO statement. Although the code which we are translating generally only uses GOTO statements to construct structured repetitions and to handle exceptions, we decided not to try to parse these GOTO statements into structured Ada equivalents. We felt that the risk of not matching the structure of the FORTRAN code was too great. We do generate a cross-reference table, showing the locations of all GOTO statements, and print it out separately from the Ada source code file.

There are a number of FORTRAN features which we flag in the output and do not attempt
to translate with the translator. Common blocks in which different variable names are used to refer to the same memory location are not processed. This is not translated, since it is not considered to be a sound structured programming practice. If the common blocks use the same variable names in all routines, then they are simply translated into PACKAGES, and treated as global memory. For the same reason that we do not allow the first type of common block usage, we do not translate EQUIVALENCE statements. The FORTRAN INCLUDE statement is also not processed by the translator. A person must copy the included files into the main file before it is processed. The FORTRAN SAVE statement, which allows the local variables of a routine to be saved in between calls to that routine is also not supported by the translator. Finally, formatted READ and WRITE statements are flagged for translation manually.

The restrictions discussed above made the design of the translator relatively simple. It is being constructed in Ada on the DEC VAX 11/780. The translator consists of three stages:

a) The Lexical Analyzer
b) The Parser
c) The Code Generator

The lexical analyzer takes the FORTRAN source code file as its input. It produces a token stream as output. Each token is a record, which contains two numbers. The first number gives the class of the token (e.g., identifier, number, symbol, label); the second number identifies the specific token (e.g., the key word IF, the integer 5). The lexical analyzer uses the first character of a lexical unit to tell it which token class it must process. It terminates processing that token when it picks up a character which cannot be part of that token.

The lexical analyzer has been designed so that it can easily be modified. The maximum allowed length of a FORTRAN identifier is defined as a constant. It is also set up to handle addition of extensions to FORTRAN, such as longer identifiers or DEC extensions to the language.

The parser takes the FORTRAN token stream as its input, and produces a stream of Ada tokens as its output. It parsesthe FORTRAN token stream into tree structures, suitable for translation to Ada. If it encounters one of the FORTRAN constructs we decided not to process automatically, it flags that entire construct in the output file, and enters a reference into the Manual Translation Cross Reference Table, which is printed in a file separate from the Ada source code.

The code generator takes the Ada token stream produced by the parser as its input, and generates an Ada source code file as its output. It also generates the Manual Translation Cross Reference Table (mentioned in the parser description) and the GOTO Locator Table (discussed in the beginning of this section).

Before the code generator translates the Ada token stream into source code, it offers the user an opportunity to replace the FORTRAN variable names with longer variable names. The limit on the length of the Ada variable names is defined in a constant. Once this substitution is accomplished, it is very simple to translate the Ada token stream into Ada source code. This is accomplished through a straight-forward one for one substitution of keywords, identifiers, numbers and symbols for their tokens.

The final stage of the code generator is a 'pretty printer.' This section of the program recognizes keywords, and performs proper indentation, in order to enhance the readability of the final source code. It also has a limit on the number of characters which may appear in any line of source code. This is defined in a constant. The pretty printer will only print one statement per line.

CONCLUSIONS

We have determined that the benefits outweigh the disadvantages of converting some—but not all—of our FORTRAN code into Ada. We ultimately enhance portability and simplify maintenance when we move toward one common language.

Our experience has shown that the construction of a simple translator is a reasonably efficient approach to the FORTRAN to Ada code conversion problem, provided that the FORTRAN source code is of high quality, when judged by structured programming standards. The investment required for the construction of this simple translator is moderate, and the potential benefits are significant.

We contend that code which cannot be easily translated into reasonably high quality Ada automatically should be redesigned and rewritten manually.

Finally, we found that the analysis, design and construction of the translator was an excellent training experience. Engineers who had worked primarily as applications programmers expanded their knowledge of both FORTRAN and Ada, in addition to learning how the front end of a compiler works.