N89-16364

SOFTWARE ISSUES INVOLVED IN CODE TRANSLATION OF C TO ADA PROGRAMS

Robert Hooi, Joseph Giarratano
University of Houston Clear Lake

ABSTRACT

It is often thought that translation of one programming language to another is a simple solution that can be used to extend the software life span or in re-hosting software to another environment.

This paper examines the possible problems, the advantages and the disadvantages of direct machine or human code translation versus that of re-design and re-write of the software. The translation of the expert system language called C Language Integrated Production System (CLIPS) which is written in C, to Ada, will be used as a case study of the problems that are encountered.

1 FUNDAMENTAL CONCEPTS

1.1 Introduction

CLIPS is a rule-based expert system language developed by the Artificial Intelligence (AI) section of the Johnson Space Center. The programming language C was used in the original implementation of CLIPS, while Ada is used as the new target language.

In re-hosting the original version of CLIPS from C to Ada, two approaches were attempted. The first approach was direct code translation, while the second was a complete re-write and re-design of the entire software.

1.2 Direct Code Translation As A Possible Approach

The work involved in the development of large software systems often represents huge amounts of time and expense. Monetary investments and time involved in the development make it extremely desirable to continue using these software systems for as long as possible. A few reasons for re-hosting to a new hardware or software environment are:

- o software system is still needed
- o difficulty in locating technical support
- o need to increase software versatility
- o greater execution speed
- o more economical hardware

At first sight, code translation may be seen as a simple, inexpensive approach to a complex and difficult problem. Translation seems to offer an attractive patch in extending the versatility and life span of existing software systems without the need to "re-invent the wheel".

1.3 Advantages Of Direct Code Translation

Direct code translation is often considered a very direct, simple and desirable method of re-hosting existing software to another environment. It offers a number of plausible advantages that can be highly questionable in retrospect. These advantages are:

- o elimination of some of the software life cycle phases
- o requires less time and effort compared to re-design and re-write
- o easily extended life span and software versatility
- o elimination of human errors compared to re-design and re-write.

These advantages will now be discussed in more detail.

1.3.1 Elimination Of Some Of The Software Life Cycle Phases

The major phases of the software life cycle [1] include:

1. requirement analysis	2. specifications/requirements
3. design	4. coding
5. verification and validation	6. maintenance and operation

If carried out correctly, the most difficult work involved in the software life cycle is in the early phases. Maintenance and operation may be time consuming but lacks the complexity of the first phases (1 - 3) of the life cycle, unless major changes are desired after the software is released. In fact, studies have shown that maintenance may account for up to 90% of costs for the software life span [8,9]. One of the major reasons for the development of Ada was to reduce maintenance costs. Direct code conversion offers a simple short cut to avoid the early phases (1 - 3) of the life cycle by:

- o requiring only source code of the software
- o minimizing verification and validation
- o allowing re-use of test data from the original
- o eliminating the need to do design conversion
- o eliminating the need to understand the functionality of the software, which is especially useful if the original programmers have left

In cases where the documents of the early phases of the software life cycle are missing or are poorly defined, direct code translation means eliminating "re-inventing the wheel". There is no need to derive a design if it is missing or to study and redefine poorly written documents.

1.3.2 Requires Less Time and Effort Compared to Re-Designing and Re-Writing

Direct code translation appears to be an attractive approach in that it is theoretically a simple, mechanical process if the host and target language are similar. There is little need to understand the actual requirements, design or functionality of the program. Much smaller machine and human resources are needed in performing the translation. All that is required is a basic understanding of the software tools and their interfaces, detailed knowledge of the host and target languages, and the different hardware specifications.

Once a translator is built, code translation may proceed very quickly and without the possibility of human error. In theory, translation work is basically tedious but mechanical and simple in nature. All that is required is a consistent, correct and accurate equating of the original implementation with that of the target language and its' environment. Also, once a translator is available, it may be used on other software or the translator may be sold for a profit.

1.3.3 Easily Extended Life Span And Software Versatility

A re-write and re-design of a software system is expensive and time consuming. It requires a considerable amount of professional human expertise compared to direct code translation which could be performed by either human or machine translators. If direct code translation is done by human translators, it may be expensive and time consuming, but it is still cheaper than a re-design and re-write. The early phases (1-3) of the software life cycle can still be skipped.

If the work is performed by a machine translator, it would still be relatively inexpensive since the only real work would be in the validation and verification of the accuracy of the results. A certain amount of editing and debugging may be required, but the work involved is relatively mechanical in nature while the resources needed are still less than an actual re-design and re-write.

The elimination of most of the work involved in the software life cycle, plus the possible availability of a machine translator and the ease involved in the work, could mean a saving in time. The re-hosting and re-targeting work can be completed in a relatively short period of time compared to re-design and re-writing.

1.4 Elimination of Human Error

Translation of computer software can be carried out either by a human or machine translator. If the software is large, then the use of a machine translator may be the least expensive approach, whereas for small programs, a human translator may be the better choice. The translated version is tested for accuracy and

correctness by computer programmers.
The requirements for a human translator is an understanding of the computer languages and external interfaces involved in the translation process. The work in general is very mechanical in nature. The advantage of the mechanical translator compared to a human is a reduction in software errors in the translation. The disadvantages of machine translators is that the human may clean up the code during translation because the human can understand the semantics as well as just the syntax.

2 SOFTWARE ENGINEERING ISSUES INVOLVED IN DIRECT CODE TRANSLATION

2.1 Introduction

The process that direct code translation generally takes often results in a failure to address certain design and implementation issues in software engineering. These can develop into major problems later on in the software life cycle. They are:

- o differences and incompatibilities in design methodologies
- o differences and incompatibilities in language implementation
- o possible disregard of the richness of the target language
- o possible inaccuracies and discrepancies between languages
- o maintenance costs may well exceed savings of translation.

Unless the above issues are addressed, the problems and disadvantages may outweigh all the advantages made in a direct code translation.

2.2 Differences And Incompatibilities In Design Methodologies

The types of available software tools have a profound effect on our thinking process and thus the design and development of the software. The types of design methodologies used are often guided as well as restricted by the software tools used in the implementation of the actual program. It should be noted, however, that methodologies are generally much easier to compromise than the implementation language tools. The same rules apply to modern software engineering principles and practice.

The principles of modern software engineering as incorporated by languages such as Ada and Modula-2 are [2,3] are:

o modularity	o abstraction	o information hiding
o localization	o uniformity	o completeness
o confirmability		

If the programming language used does not directly provide support towards the above software engineering principles, then it is difficult and often impractical to implement a design which adheres to these concepts. The implementation language and design methodologies used should be mutually compatible for best results.

So converting a BASIC program to FORTRAN IV would be reasonable since they share similar software principles. Likewise conversion from Modula-2 to Ada would be feasible since both languages support the above modern engineering principles. Difficulties arise in the translation of programs in a language like C to Ada since C does not adhere to the above principles of Ada.

The use of an object-oriented design methodology [4], together with an object oriented language such as Ada, forms a very highly compatible choice towards the support and implementation of these software engineering principles [10]. If a programming language does not readily support these concepts it will probably be absent in the implementation. In languages which do not have this support it may be too expensive and difficult to enforce these principles. In most cases, the designers and implementors would probably choose a design methodology that allows an easier implementation, rather than one in which the language would have difficulty adhering to.

A major issue involved in directly translating a program to a target language is that the type of methodology used is often ignored. If the work is performed by computer programmers, then it may be possible to modify and adapt some of the code to that of Ada's object-oriented approach. It would be impossible for a simple syntax-directed machine translator to do this completely, since it involves a certain degree of independent thinking, analysis and understanding of the original software. Thus, a machine translator would have to understand the semantics as well as the syntax to do a thorough job. Such a translator would have to include artificial intelligence and expert system techniques and would be very difficult to build. A simpler alternative would be to have a human examine the code produced by the simple translator and polish it up. However, this could still be a major task.

If the original implementation is not an object-oriented design methodology, then it will not normally be present in the translated version. For example, if the original does not support the concept of information hiding, then the translated version will not. If the original design methodology adheres to the concept of data flow decomposition or the Jackson Design methodology [5], then the translated version certainly would not have any of Ada's object-oriented approach.

A time factor should also be taken into account since the type of methodology used is dependent on when it was first conceived. Ada's object-oriented design methodology would certainly be absent if the software was developed prior to the 1980's. This technical gap may not be easily bridged in direct code translation unless the languages are similar, such as Modula-2 and Ada.

The ability of the language to support these methodologies must also be considered. For example, Ada's packages supports the concept of information hiding, which may be simulated by C's statement "INCLUDE". However, this does not mean that C provides the same capabilities or support of the concept of information hiding found in Ada. There is no close equivalent in C to Ada's private and limited private types or visibility controls.

Translation becomes even more difficult concerning the concept of re-usable software components. For example, there are no facilities in C to directly simulate Ada's generics.

A major difficulty in translation occurs when the documentation is missing. The problem is compounded when the methodology used is unknown and is not similar to Ada's object oriented approach. These problems were found in the translation of CLIPS.

A certain amount of re-design and re-write was required in certain program segmnents in order to conform to the language implementation requirements of the target language (ADA). An example is the difference between a C library program versus Ada's packages. Each C library program has the function - "main", which may make calls to other external library functions or functions within the same file. The visibility rules in C allow calls by the sub-program unit "main" to other functions located anywhere within the file dependent upon the programmer's convenience. Ada's visibility rules allow procedures and functions to be called by other program units only if declared above it. An example found in the CLIPS demonstrating C's visibility problems is shown below:

```
main ()
   {
         :
      command_loop ();
      if (opt_u_found == TRUE)
         {displayfunctions();}
     }
      :
command_loop ()
      :
displayfunctions ()
```

Ada's visibility rules would require:

```
procedure Command_Loop is -- assumes converted to a procedure --
     :
end Command_Loop;

procedure Display_Functions is
     :
end Display_Functions;

procedure Main is
   :
begin
   :
   Command_Loop;
   if ( Opt_U_Found = True ) then
      Displayfunctions;
      :
end Main;
```

In view of the differences in design methodologies, it follows that if the translation does not include the methodologies, then the work is only partially complete. A translation without the design methodology is not a true representation of the target language's environment. It is therefore not possible to re-target software correctly by direct translation if the design methodologies are not considered in the work.

2.3 Differences And Incompatibilities In Language Implementations

Discrepancies and incompatibilities between different computer languages mean that what is considered as an acceptable programming practice in one may not be permitted in another. C has weak typing, which means that unless it well enforced, most data types can take on any values assigned to them. If the program is to be properly translated to Ada, then a number of conversions and data checks must be included to restrict the values assigned to variables. This is needed to accomodate the differences between C's weak typing versus Ada's strict typing requirements.

The strength in Ada's requirement for strict typing enforces program reliability and consistency, while C allows for greater flexibility on the part of the programmer. The result in accomodating the typing requirements of Ada is that the translated version is seldom, if ever, smaller than the original. In the translation of CLIPS to Ada, it was found that for every line of C code, the average is generally two lines of Ada code. This does not mean that Ada is a less efficient language compared to C, merely, that Ada's strict typing enforces consistency and provides a more reliable program. This is particularly important to the Space Station since much of the software will support human lives and also directly affect the longevity of the space station.

An example from CLIPS demonstrating C's weak typing which must be corrected in the translated version is shown below:

```
    :
  float tally = 0;
    :
  char lm;
    :
  int ten = 10;
    :
  tally = tally*10 + (lm - 'o');
```

The Ada version must have the following changes made:

- o convert integer 10 to float
- o convert data types: lm and 'o' to ascii values
- o convert the resulting arithmetic operations (lm - 'o') to float
- o value initialized to tally changed to 0.0

```
tally := tally*10.0 +
   float(Character'Pos(lm) - Character'Pos('o'));
```

The complexity of the problem increases if the typing problem occurs in the arguments of a subprogram call. Data conversion will have to be made prior to actual passing of the values to the subprogram call.

In addition to the weak typing problem, certain language features in C which are not found in Ada have to be worked around. This again accounts for some extra code being produced. An example from CLIPS showing the auto increment is:

```
while ((atemp != null) &&  (++count != nnn))

           versus Ada's version

while ((atemp /= null) and (count /= nnn)) loop
   count := count + 1;
           :
end loop;
```

Note that in this case, the lack of an auto increment or decrement in Ada does not necessarily mean it is a slower language at run time. Depending upon the compiler implementation, the functionality is the same and should execute at similar speeds. The major difference is that Ada aids readability, thus making it easier to understand and maintain.

The extra code size may present several important problems:

o program efficiency could be sacrificed

o storage and execution speed becomes worse

o maintenance problem increase due to increased code size

Depending on where the increased code is generated source code, code size could result in slower program execution. In a situation where response time is crucial, such as real time execution, anything that may reduce execution speed should be examined very carefully to see if it could be acceptable.

For software systems that are relatively small, an increase in size may not pose an important issue. However, as the magnitude and complexity of the software increases, there will be a proportional hardware demand. For example, consider a large embedded software program occupying 100,000 blocks of disk space. Increasing the code size by two times might exceed the remaining disk capacity. If this rule is applied to software systems that are even larger, then size requirements made by direct code translation may not be an acceptable solution.

An increase in code size would also mean that the complexity of software maintenance would also grow. Issues in software maintenance will be further examined later in this paper.

Some of the results found in the translation of CLIPS to Ada:

Comparison of storage size for one of the files on the VAX:

original version:	CLIPS.C	occupies 175 blocks
translated version:	CLIPS.ADA	occupies 369 blocks

Comparison of code size for functions:

Excluding global data declarations, for function Rarray
the original occupies approximately 15 statements
Excluding global data declarations.
translated version occupies 26 statements.

Some factors contributing to an increase in code and storage size are:

- o statement terminators found in Ada - end if, end case and others
- o instantiations of generic I/O packages
- o path names used in calls made to other packages
- o absence of auto increment and decrement statements
- o absence of statements with embedded functions and statements in boolean tests, such as auto increments

Additional explanation of the reasons for the increase in code and storage size will be discussed in the next section.

2.3.1 Possible Disregard Of The Richness Of The Target Language

In order to translate as accurately as possible, the simple syntax method is to equate statements found in the original with that of the target language. This presents a disadvantage in that much of the richness found in the target language is often ignored. If the translation is done manually, then certain segments of the original could be re-built to allow better usage of the target language. The same cannot be easily applied if the work is performed by machine translators unless semantic understanding is also included.

Ada has a standard of 63 reserved words regardless of the implementation versus C's approximate 33 (including functions for the C preprocessor). These 33 words of C depend upon the compiler, version and host environment. Ada has, in addition, a number of features which are not present in the standard C implementation. They are:

- o predefined language attributes
- o predefined language pragmas
- o predefined language environment:
  - o language predefined identifiers (package standard)
  - o utility packages such as system and calendar
  - o input and output packages
- o ability for overloading, generics, multi-tasking, nested generics and packages

The use of generics would drastically reduce the amount of code found in the original, since functions with like actions but different data types and properties can be grouped together and placed in the same subprogram. As a generic unit, a template is built to accommodate the function of a sub-program without specific properties. The instantiation allows the properties to be set to the generic package.

While C allows for greater flexibility in usage, the richness of Ada permits better control, reliability and flexibility in the programming environment. For example, in order to recover from run-time errors, the C program will have to simulate what Ada naturally does in its ability to raise and handle exceptions. A direct syntax translation would result in having a simulation of run-time error recovery in Ada, which ignores what the language is equiped to perform naturally.

An example taken from CLIPS is:

```
if (notstate == 0)
   {
    if (btemp == NULL)
       {
        htemp -> locals = valuescopy(line->locals);
```

If the power of Ada is exploited correctly, then the structure above could be combined, yet simplified as follows:

```
if ( Notstate = 0 ) and then ( Btemp = null ) then
   Htemp.Locals := Valuescopy ( Line.Locals );
```

It should also be noted that since Ada data types are not case sensitive, then capitalization of the variables could be used to improve readability, and so provide better maintainability.

The full power of the target language is seldom exploited completely in direct translation. The increase in code and storage size of the translated version is in no way an indication that the original host language is a better software tool. The same rule applies if the execution speed of the target language is reduced. It does not mean that Ada is a less efficient language, merely that it is not exploited fully.

2.3.2 Possible Inaccuracies And Discrepancies Between Languages

Translation of language syntax is generally a very mechanical process. To equate accurately, it necessary to consider the semantics of a program, which is a much more difficult task.

The difficulty of the problem of correct semantic translation increases with the magnitude and complexity of the software. In addition, if the source in the original is poorly written and has a very confusing implementation, the chances of a misinterpretation increases. The main software issue is the program's reliability, accuracy and correctness. If the semantics are misconstrued in a subtle area that is difficult to detect, then locating and debugging the logic problem would be equally difficult.

The differences and restrictions in language implementation are a major cause of discrepancies in translation. For example, C permits recursion for the arguments in a function call since the values passed into the function can be changed. In contrast, the parameters in Ada must be of a formal type, and changes to those values are not allowed. To work around this problem, the translator must decide whether to declare the values that are changed in the function as global data types or convert it into a procedure. If the values are changed into global data types, then the issues of localization and modularity are raised. In addition, care must also be taken to ensure that those global values are correctly initialized, changed or kept at each call. If the values are not traced correctly, then the program execution may not function as originally designed or there will be a set of global values created at every subprogram unit that makes a call to that function. If a function is converted into a procedure, then the calling process made by the subprograms will have to be changed.

An example from the CLIPS demonstrating the changes made to the arguments in a function as follows:

```
Any(code, values)
int code;
   :
    if (values->whoset == code)
        ret = -1;
    else
       values = values->next;
   :
   return(ret);
```

Note that in C the arguments of a function are value parameters. It can, however, perform as a variable, formal or value parameter. Ada strongly enforces the type of parameter used, which is defined in the subprogram arguments. For example:

```
push ( first, second, third )
```

versus Ada's parameters

```
procedure Push
   ( First  : in Integer;
     Second : in out Float;
     Third  : out    Boolean );
```

This ensures program reliability and consistency, as values passed in are restricted to performing within the scope of their declared type. The simple solution in translating from C to Ada is to have all arguments declared as value parameters. In translating the C code to Ada, unless checks are made to determine if the arguments passed perform as a variable, formal or value parameter, this particular strength in Ada will be ignored.

Another possible semantic problem in direct translation is that C is a case-sensitive language. A data type with the same name but written in upper-case is a different variable to that which is in lower-case. Caution must be taken to ensure that data types with the same names but different cases be given different names. In addition, variables in C may be reserved words in Ada. The translator must be able to identify these and assign meaningful substitutes.

Examples taken from CLIPS to the problem above is shown below:

```
struct element *out;     extern struct internode *AGENDA;
      :                                        :
while (out != NULL)     struct internode *agenda,*step,*past;
      :                                        :
```

```
    out = out->next;                  AGENDA = agenda;

          (1)                               (2)
```

In example (1) a compilation error would result if the translation process does not substitute a different name to the data type - "out". The data name "out" is an Ada reserved word and cannot be used as a variable name.

Example (2) can have unpredictable results, depending on the translated version. Since Ada is not a case sensitive language, the translated statement could really be doing nothing, unless a change is made to either one of the two object names - "AGENDA" or "agenda".

The ability of C to include function calls in test statements further complicates the translation process since a patch must be used to adapt to Ada's language requirements. Temporary variables must be used in order to obtain the values required for the boolean tests prior to the execution of the statements. Again, the issue is not that Ada is a less efficient language, but that it enforces program readability for better maintainability. An example from CLIPS shows the problem:

```
    if ((any(go,list) == -1) && ((second == -1) ||
       (any(second,list) == -1)))
```

while an Ada patch solution would be:

```
            :
  First_Value, Second_Value : Integer := 0;
            :
  First_Value  := Any ( Go, List );
  Second_Value := Any ( Second, List );
            :
  if ((First_Value = -1) and
     (Second = -1 ) or
    (( Second_Value = -1))) then
```

Bit manipulation [6,7] is another area that Ada does not <u>directly</u> support, but is present in C. The translator must be able to use an Ada implementation of the compiler that can perform a representation of the size of the bit used. There are also bit manipulation operators in Ada similar in C. A patch must be found in order to translate correctly and accurately. Note that this problem did not arise in the translation of CLIPS as there was no bit manipulation used.

In certain cases it may not even be possible to implement a direct translation. For example, problems arise when the original implementation performs systems calls using operating system dependent control languages such as IBM JCL,DEC BLISS and DCL. Problems occur also when the target language does not contain the necessary interface features. Direct code translation is thus dependent upon the implementation capabilities of the target language and its host environment.

<u>2.4 Maintenance Costs May Well Exceed Savings Made in Translation</u>

The quality of the simple syntax translation is at best equivalent to the original. In most cases it is inferior to that of the original. The reason is that direct translation copies over the raw design and implementation of the original. If the source code in the original is unstructured, cryptic and consists of meaningless data names, then the translated version would bear the same resemblance. As the old saying goes, "Garbage in, Garbage out". Just because a program is translated to Ada does not automatically make it a good program. In addition, the increase in code to patch some of C's weak typing serves only to complicate the task of maintaining the software.

In any language, if the original has poorly designed code, it will be hard to translate or re-design correctly. Ideally, the person who wrote the original code should also have been trained in the target language. For example, the best person to write C code would be an Ada programmer who knows C. A programmer who has experiences with a more evolved language such as Ada, will write better C than one who knows just C.

An example an equivalent translation from CLIPS is:

```
if (((element->state == 'o') || (element->state == 'n')) &&
   (element->type != FCALL) && (element->type != COAMP ))
   {
  while ((list != NULL) && (stop == 1 ) && (go == 1))
     {
    if (element->name == list->name)
       {
       go = 0;
       pkg = 0;
       if (element->type == NUMBER)
          if (element->ivalue != list->ivalue)
             stop = -1;
                :
```

The translated version in Ada is shown following:

```
if (((element.state = 'o') or (element.state='n')) and
    (element.type /= FCALL) and (element.type /= COAMP )) then
          :
 while ((list /= null) and (stop = 1) and (go = 1)) loop
    if (element.name = list.name) then

       go := 0;
       pkg := 0;
       if (element.type = NUMBER) then
          if (element.ivalue /= list.ivalue) then
             stop = -1;
                :
```

Note that the code and structure characteristics present in the original can also be found in the translated version. These are:

o meaningless, cryptic object names

o lack of capitalization standards for readerability o poor structures and language usage

Code from CLIPS that has data types with meaningless names like jill, jack, junk, grab and has no documentation, will produce a translated version with the same characteristics.

In the simple syntax translation of CLIPS to Ada, the general results obtained were that the translated version was worse than the original. It is at best an Ada program written in C methodology, with Ada structures looking like C structures. A simple syntax translation of bad C code will produce bad Ada code. However, this does not mean that good C code will produce good Ada code, since much of the wealth of Ada is ignored. This defeats the purpose of the translation to Ada, which is supposed to improve maintainability and reliability. If the software lacks quality, it cannot be easily built on, understood, modified and most important - maintained.

2.5 Summary

In view of today's rising software costs, where the bulk (80% - 90%) of the expense lies in maintenance and operation, direct translation may not be the best alternative in extending the versatility and life span of a software system. It is at best a patch and at worst an expensive solution when maintenance is considered.

3 DIRECT TRANSLATION VERSUS RE-WRITE AND RE-DESIGN OF COMPUTER SOFTWARE

3.1 An Evaluation Of Direct Translation

Simple syntax code translation may not be the ideal solution to a difficult and complex problem. Yet it is not a totally useless approach since there are certain values that are tied to the process. For example, if the program is relatively small, simple, and has a limited life span and usage, then translation may well be the best approach. In certain cases, where a design document is not present, a translation may be a possible method used to build a prototype for study purposes prior to the actual re-write of the entire software.

Because of the expense involved, human translation is generally limited to small and simple programs. The cost of human re-writing and re-design is best served in real-time code where performance is critical. In the translation of CLIPS, it was found that the time spent in the translation process was almost equivalent to that used in the original implementation and was thus self-defeating.

Simple syntax translation should be avoided if the target environment has a very different design methodology. It can be strongly considered if the target is a different host machine or a new version of the same language and design methodology.

3.2 An Evaluation Of Re-Design and Re-Write

The re-design and re-write approach should be strongly considered if a well documented design along with the specification exists. The original types of methodology used may be incompatible with the target , but it may be converted and adapted to the requirements of the new design methodology. The reason is that if the design is clear and well documented, it can be easily understood, worked upon and modified to fit any methodology. Coding is a relatively simple and mechanical process if a good design exists. The most difficult work involved in the development of any software is still the early phases of the life cycle. If a design exists, it can be studied, the weaknesses can be avoided in the implementation, and the strengths enhanced further.

The only part of software that can be transparent to all languages and host machines is the design and its' specifications. Once the design is converted to suit the requirements of the new methodology, it can be ported to the new target language and host machine. "Re-inventing the wheel" can be avoided only if a design is present.

3.3 Summary

Simple syntax translation or re-design and re-write are alternatives that can be used, but these have to be carefully considered before either one is adopted. Re-design and re-write should be strongly considered if a design is present. Translation may be considered if the goal is to port the software to a different host machine or up-date the software.

Considering the fact that neither of the two approaches is exactly easy to adopt, a few possible alternatives can be taken into account. These are:

- o interface the original with Ada
- o implementing the new software in Ada and port the data produced by the old software to be processed in the new environment

Interfacing Ada with other languages can be done using the Ada language's predefined pragma INTERFACE. Consideration should be given to the possible restrictions due to the different implementations of the compile. The reason being that this pragma is an implementation feature dependent upon the Ada environment. Certain implementations may allow for full usage, while others may be used partially and some none at all [6].

In cases where the host hardware is out-dated and an Ada compiler may not be available or an interface with the target language cannot be made, then it may be advisable to use the old software to generate the data. Any additional processing that is not dependent on the old software may have the new implementation developed in Ada. The data generated can be ported and executed in a new host environment.

Conclusion

Direct code translation or re-write and re-design may not be the only available solutions. There are basically no cheap and easy solutions to the problem. In terms of today's need to reduce the cost of software maintenance, plus the greater importance of software reliability, it may be much better to rebuild the entire system correctly. The advantage is that the faults and weaknesses are known and can be avoided. A better, more reliable software system can be built in place of the original.

Acknowledgements

Support for this work was provided by NASA Contract NAS9-17010, Task No. B13.C

References

1. Giarratano, Joseph, Foundations of Computer Technology, pub. by Howard W. Sams, 1982.

2. Douglas T. Ross, Hohn B. Goodenough, C.A. Irvine, "Software Engineering: Process, Principle, and Goals" Computer, May 1975.

3. B. W. Boehm, J. R. Brown, M. Lipow "Quantitative Evaluation of Software Quality", Proceedings of the Second International Conference on Software Engineering, pp. 592-605, 1976.

4. Grady Booch, "Object Oriented Development" IEEE Transactions on Software Engineering, Vol. SE-12, No. 2 February 1986.

5. J. R. Cameron, "Two Pairs of Examples in the Jackson Approach To System Development" Proceedings of the 15th Hawaii International Conference on System Sciences, January 1982.

6. Donald G. Martin, "Non-Ada to Ada Conversion", Journal of Pascal, Ada, & Modula-2, Vol.4, No.6, pp. 36-40 (1985).

7. Douglas L. Brown, From Pascal To C - An Introduction to the C programming Language, Wadsworth Publishing Company, 1985.

8. Girish Parikh, The Guide to Software Maintenance Winthrop Publishers, Inc, 1982.

9. Ian Sommerville, Software Engineering Addison-Wesley Publishing Company, 1985.

10. Mark W. Borger, "Ada Software Design Issues" Journal of Pascal, Ada, & Modula-2, Vol. 4, No.2, pp. 7-14, 1985