System Programming with Ada

K. Hosokawa
Science Institute
IBM Japan

INTRODUCTION

Software maintenance is one of the difficult and costly stages in the software cycle. To maintain a program, one must first understand it. This is not a simple task and is often difficult even for the experts.

The difficulty in understanding a program has some relation with the characteristics of the program itself. The characteristics may be of the following.

size It is more difficult to understand a big program than a small program.

complexity The complexity of the program is related to the complexity of the algorithm which it implements.

language Certain languages are suitable for certain applications and thus programming with an ill suited language reduces readability.

Considering operating system (OS) programs, its size is huge, complex, and usually written in assembler. To maintain such programs is no easy task. However one is faced with this problem and some how must come up with a solution.

This paper describes one such attempt. Here the language characteristics of OS programs is tackled. The program which is written in assembler is transformed into a higher level language. In doing so the program structure is easier to understand hence should improve readability.

COMPARISON OF ASSEMBLER AND ADA

A comparison of assembler and Ada is made from the view point of readability.

Readability of programs is enhanced by good structure. This implies the restricted use of 'GOTO's.

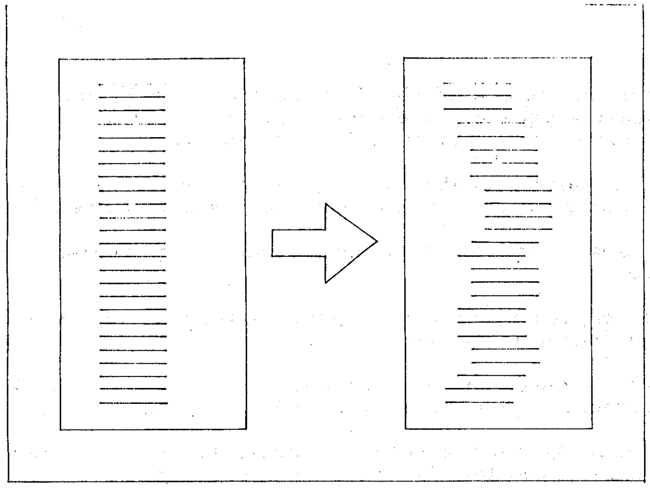
In assembler, the use of 'GOTO's can not be avoided. Even worst, there are no strict rules governing the use of 'GOTO's and hence programs turn into what can be called 'spaghetti' programs. In Ada, however, although 'GOTO's are allowed there are rules governing its use and thus 'spaghetti' programs can be avoided. Structuring constructs such as 'if', 'while' and others are also available.

The subroutine construct of assembler allows modularisation of programs. This however is not as powerful as the 'procedure' construct of Ada. For example, parameter passing is not the concern of the user of Ada. Ada also has the 'package' construct which allows data abstraction.

Thus to write structured programs in assembler one must be careful and disciplined since no check can be made by the assembler to detect nonstructured programs. Although an Ada compiler can not check for nonstructured programs, it can guide the programmer to discipline him/herself.

CONTROL FLOW TRANSFORMATION

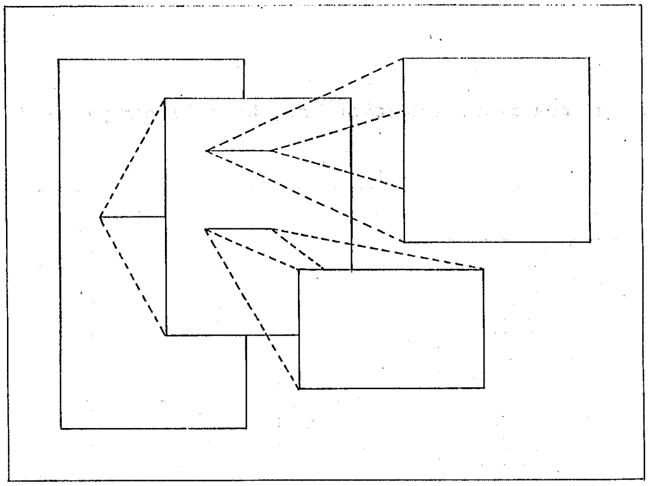
It is difficult to visualise the structure of an assembler program. The listing is one dimensional, in other words top to bottom and no left to right movement. The structure can be elaborated by simply indenting the 'branch' operators. The result can be seen in the following figure.



In this way, it is easier to see how the control flows. Now the listing is two dimensional in the sense that their is movement in both top-down and left-right manner.

By simply indenting, what is actually done is to convert 'branch' instructions into a higher level construct. Depending on where the 'branch' instruction branches the construct can be an 'if', 'while' or other loop constructs.

The idea of extending the dimensionality of the listing to aid the visibility of the structure, one can allow a third dimension. The third dimension is a movement in and out of a listing. This can simply be achieved by the current bit map display technology. This third dimension is used to visualise subroutine calls. The body of a subroutine is usually in the same level, ie. before or after the main routine, but by placing it on top of the call will enlight the hierarchical structure, viz.



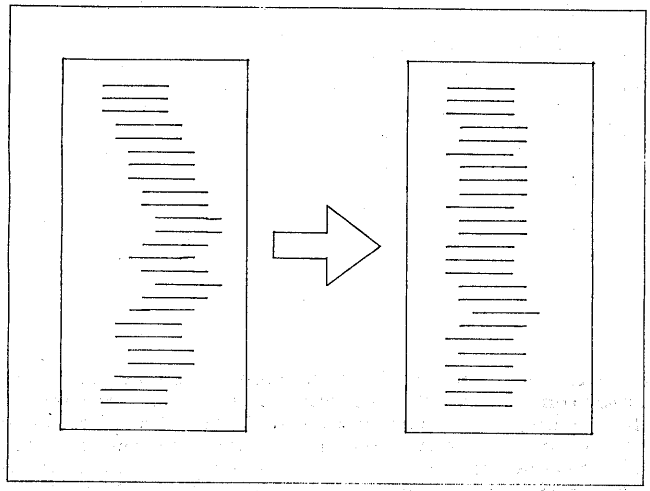
Once indentation, via 'if' and other statements, is added to assembler programs, new information about the assembler program emerges. Sometimes the nesting of 'if-then-else' statement can be so large that the structure is again difficult to visualise. In some instances this problem of can be solved by transforming 'if' statements into 'case' statements. This transformation is possible when the condition expression of the 'if' statement checks the same variable. So for example,

```
if A = 0 then
  B
else
  if A = 1 then
    C
  else
    if A = 2 then
      D
    else
      E
    end if
  end if
end if
```

transforms into

```
case A is
  when 0      => B ;
  when 1      => C ;
  when 2      => D ;
  when others => E ;
end case ;
```

In a larger scale, the transformation has the following effect.



The nesting is now reduced to a minimum.

The modularisation of the assembler program can now be attempted. One way to modularise is by grouping repeatedly used sequence of instructions into a single procedure call. This method will allow frequently used operations to be collected and transformed into higher level instructions. Some examples are incrementing variables, accessing an array and so on.

The above method of modularisation is simply a grouping of similar patterns. Modularisation shows its effectiveness when the procedures consist of logically connected instructions. To determine whether a sequence of instructions are logically connected and grouped into a procedure requires human intelligence. Without some sort of intelligence this can not be achieved.

However there are instances where modularisation of logically connected instructions can be made without human intelligence. The sequence of instructions after a condition of an 'if' statement may be grouped together in a procedure. The reasoning behind this is that starting from a whole program by executing the 'if' statement, one is sieving the possible instructions that can be executed. Thus when a group of instructions is to be executed after several 'if' statement sieves, these instructions have a logical connection.

This method of modularisation will generate modules which have a single logical meaning, but to subdivide such a module into further modules requires human intelligence.

From the flat assembler program, methods to incoorporate structured constructs and to modularise so as to generate structured programs have been suggested. Using these methods one can visualise the structure of assembler programs more clearly.

DATA STRUCTURE TRANSFORMATION

The control flow transformation applies to general programming. Transformation of data structures are however difficult to apply to general programming since usually data structures are diffused thoughout the program. In OS programs, this is not the case. OS programs manipulate hardware which can be seen as data structures. These data structures are static in the sense that their structures do not changed.

Since these data structures remain static, they can be converted to abstract data types or Ada 'packages'.

Ada 'packages' allow the encapsulation of data and restricted use of that data. Allowing only restricted operations to a data is useful when controling access. For example, a user program should never be able to alter the program status word. By using a 'package' to cover the program status word and allow only certain instructions, the system is protected from user misuse.

These data structures can be collected from the assembler data definitions. The data definitions are used to define the size of a variable or define constants. A collection of variables can be made into an Ada 'record' and then 'package '. The logical connection to determine which variable definitons should be grouped into a single record does not require human intelligence. Since the data definition in question is usually a direct mapping of the hardware, the information in the assembler program is enough to generate a 'record'. This however may generate a flat 'record' definition with no structure.

The operations that are allowed by the data usually can be deduced from the type of the variable.

When the variable has a type of one bit, it is used as a flag. The operations available to a flag is to test its value, to set or to unset it. This one bit variable can be replaced by a boolean variable. For example,

```
FOUND ds 1b   -- Define 1 bit of storage labeled FOUND
```

is tranformed into

```
FOUND : BOOLEAN
```

When a variable has a type of a byte or less, it is used as a collection of flags. If a bit in the variable is set implies that a certain condition is attained out of several possiblities. This variable may be converted into a enumerated type variable. For example,

```
DIRECT ds 2b     -- Define 2 bits of storage labeled DIRECT
UP     dc '00'b -- Define constant binary '00' labeled UP
DOWN   dc '01'b -- Define constant binary '01' labeled DOWN
LEFT   dc '10'b -- Define constant binary '10' labeled LEFT
RIGHT  dc '11'b -- Define constant binary '11' labeled RIGHT
```

transforms into

```
type WHERE is ( UP, DOWN, LEFT, RIGHT ) ;
DIRECT : WHERE
```

Other variables may be interpreted as addresses, integers and so on. No rules to distinguish them is found yet.

These operations can be combined with the procedures that have been generated in the control flow transformation. If similar operations are found then the procedures can be replaced. This allows the protection of the data from misuse and also improves readability.

The techinique to transform data definitions into Ada 'record's can be seen as a discompiler. If operations performed during compilation can be reversed and applied to the assembler data definition a higher level representation of the data structure can be generated. By 'packaging' these 'record's, they can be protected from misuse.

CONCLUSION

An attempt to use Ada as a systems programming language has been described. The approach taken is different in the sense that the software is not developed from scratch, but transformed from existing assembler programs. This thus is also an attempt to tackle the software maintenance problem.

The transformation from assembler to Ada results in a clearer structured program. This should enhance maintainability.

The transformation can not be fully automated due to the lack of the transformation rules and artifitial intelligence may provide the answer.

The full Ada constructs have not been exploited. The use of tasks to describe concurrency for example between a real device and its control unit has not been investigated.

BIBLIOGRAPHY

- Akiyama, Y., 'Fuctional Path Programming', TR 03.171, Santa Teresa Lab., San Jose CA, Dec. 1981
- ANSI, 'Ada Programming Language', ANSI/MIL-STD-1815A, Ada Joint Program Office, Washington DC, Feb. 1983
- Guttag, J., 'Notes on Data Abstraction', Nato Summer School, Jul. 26 - Aug. 6 1978