Suggested Eight-Step Iterative Process for Structuring FORTRAN Modules

Ronald C. Wackwitz
COMP-AID
513 Old Bear Creek Road
New Braunfels, TX 78132

Original version: April, 1978 A.D.
Revised: November 10, 2006 A.D.

Contents

List of Figures

1 Introduction

This COMP-AID brochure presents an outline for transforming an unstructured FORTRAN module into a set of independent Functional Blocks, in which the interrelation of the <event>'s[1] associated with each of the Functional Blocks becomes highly visible. Benefits of this transformation include not only the expected ease of maintainability, but also the ease with which the present module algorithm can be improved, often with an increase in execution efficiency.

The process is an interactive one, usually requiring only two to three cycles to successfully complete. Moreover, the process is also a thoroughly rigorous one; once you have employed it, you will realize that there is nothing "magical" to either achieving working programs or to verifying their correctness. Rather, it is a simple matter of *organization*. By breaking down the code into blocks small enough to permit verification of each individually, and then building up your module in terms of hierarchical groups of these individual blocks, allowing only *proper* `GOTO` branches[2] in the process, the correctness of the entire module then follows.

The eight steps are presented in the following section.

2 The Eight Steps

1. A preliminary "clean-up" of the code prior to the actual structuring process consists of only two easily performed steps, and yet simplifies the effort involved in the subsequent structuring:
   - Delete all unreferenced statement numbers.[3]
   - Convert those Arithmetic IF's, in which reference is made to the immediately following line of code, into their Logical IF counterparts.[4]
2. Determine the series of single function steps required to accomplish the module task.
3. Segment the code into blocks which reflect the series of steps outlined above in Step #2. Place the appropriate Block Number at the front of each block, in such a format as you have defined in the Block Number Descriptor (described in Chapter 7.34 of our *RENUMF User's Manual*).
4. Functionally renumber the module using the BLOCKID renumbering capability of RENUMF.

[1] The <event>, as explained on page 19 of our revised Positional Paper, *Programming in Standard FORTRAN by Functional Blocks: A Rigorous* Structured *Approach*, is the logical condition which must be true in order for entry to be made into the related Functional Block.

[2] For a description and illustration of *proper* `GOTO` branches, please see Section 7, along with Figure 3, of our *Overview of the Methodology of Functional Blocks* brochure.

[3] Unreferenced statement numbers tend to clutter up the code. If unreferenced statement numbers are permitted to stay, then the programmer must waste valuable time continually turning back to the cross-reference to determine which statement numbers are referenced, and which ones aren't. Get rid of them once and for all, thereby simplifying the effort required in the subsequent structuring process.

[4] Logical IF's more visibly demonstrate the relational aspect of each test. For example, the statement,

```
IF (I GT. J) GO TO 20
```

is much easier to comprehend at a glance than the related Arithmetic IF,

```
IF (I-J) 10, 10, 20
10 ....
```

And this, of course, is a major point behind structuring: increasing the visibility of the inter-relations. Interestingly, the above Logical IF-also generates equivalent or better code than its Arithmetic IF counterpart.

5. Note any improper GOTO branches reported by RENUMF. Circle any listed in *red*.
6. If no improper GOTO's are found, then go on to Step #7. Otherwise:
   (a) Determine more exactly what is really happening in a block which was supposedly of a single function, but which has an improper branch into its midst. Reconcile this for each major block so affected.
   (b) Go back to Step #4.
7. Once the Level I structuring is complete, only a fine-tuning process remains.[5] The following four steps constitute the first part of this process:
   (a) Examine each of the one-line comments, of the form[6] <*event*>: *process-description*, which precede each Functional Block, to insure that they are correct.[7]
   (b) Carefully examine each of the Level 1 major blocks to determine if the process-clarity can be increased by in turn segmenting any of these into Level 2 sub-blocks. If so, then Steps #2 to #6 should be repeated[8] for these newly introduced Level 2 sub-blocks. (Of course, once all Level 2 blocks are free of improper `GOTO`'s, the analyst may want to further decompose some of these into Level 3 sub-blocks, again necessitating that Steps #2 to #6 be repeated; etc.)
   (c) Look for chains of adjacent blocks[9] to determine if process-clarity can be increased by combining two or more blocks in the chain together.[10]
   (d) As the structuring process proceeds, with the result that each step of the module task is becoming increasingly elucidated, you will now be in a position to spot

[5] The process of structuring module code into a set of Functional Blocks (in which accordingly no improper GOTO branches will exist) does guarantee on the one hand two accomplishments: (1) that the set of Functional Blocks chosen are highly independent one from another, and (2) that the inter-relation of their associated <*event*>'s becomes highly visible within the specifies hierarchical structure. Yet, on the other hand, this act of eliminating improper `GOTO`'s cannot *of itself* resolve certain other design-related questions, e.g.

- are the one-line comments preceding each block correct?
- could process-clarity be increased by in turn segmenting any given-level block into sub-blocks (hierarchical decomposition)?
- could process-clarity be increased by chunking together two or more adjacent given-level blocks (hierarchical composition)?
- do identical lines of code appear in two or more blocks, which could be separated out, using permissible structured techniques?
- is the present algorithm (i.e., module task) correct?
- can the present module be improved?
- could the Functional Strength of the module be improved by separating it into two or more distinct modules?

[6] Please see page 19 of our revised Positional Paper, *Programming in Standard FORTRAN by Functional Blocks: A Rigorous* Structured *Approach*.

[7] It is especially important to insure that the <*event*> is correct for each block concerned. Insuring the correctness of these <*event*>'s simultaneously accomplishes two other tasks:

i. it clearly *elucidates* the present module task,
ii. it permits *verification* of the present module task.

[8] With experience, the process of segmenting a module into Level 1 major blocks, and then segmenting some of these major blocks into Level 2 sub-blocks, etc., *can be combined* into one step. However, initially it is conceptually more straight-forward to proceed in a step-wise fashion as we suggest here. Moreover, since any given-level decomposition contributes more to program clarity than does the level *below* it, then even the experienced user of Functional Blocks may first want to insure that all Level 1 blocks are *correct*, for example, before proceeding to the decomposition of some of these into Level 2 sub-blocks, etc.

[9] A chain of adjacent blocks is a series of two or more blocks in which -

- the level of each block in the chain is the same, and
- branching from external sources can occur to no block in the chain except to the first one.

[10] Although a hierarchical decomposition is extremely useful in elucidating the module algorithm, yet the process can be carried to the extreme. In such cases the algorithm, rather than being elucidated, tends to become obscured by a mass of detail. Interestingly, in these cases the process-clarity can be increased by chunking together appropriate adjacent blocks in a given chain.

given sequences of code[11] which occur in two or more blocks. By separating out such identical sequences, using any one of three permitted techniques,[12] not only can the module size be reduced, but — even more importantly — many times the process-clarity can also be *increased*.

8. At this point, because each block

   (a) performs a single function,

   (b) is entered under a unique, specified logical condition,

   (c) is preceded by horizontal descriptor line(s) which visibly define its hierarchical level, and

   (d) contains statement labels which start with a multiple of its block number,

   you will accordingly note that the module task is highly visible and clearly elucidated. As a result, because the complex inter-workings of the module task are so clearly laid out before you, you will now be able to perform three final tasks:

   (a) verify whether the present module algorithm is correct,

   (b) look for ways in which the present algorithm can be improved, and

   (c) determine whether the *module strength*[13] can be improved by segmenting the module into two or more distinct modules.

3 An Abridged List of the Eight-Step Iterative Process

The abridged list of the "Suggested Eight-Step Iterative Process for Structuring FORTRAN modules Modules", shown in Figure 1 on the following page, significantly enhances the view of the eight steps, since all footnotes have been removed.

[11] Obviously, we are talking about *non-trivial* sequences of code; i.e., a sequence longer than just several lines. In the case of such *trivial* sequences, however, it is usually better to leave them in their original blocks, despite duplication.

[12] There are three techniques by which identical sequences of code can be separated out in a structured fashion:

i. If the sequences all exit to a *common* entry point, then simply place the sequence in a block of its own, located just prior to the block containing the common entry point, and at a hierarchical level greater than or equal to that of the highest level block originally containing the sequence, but less than or equal to that of the block containing the common entry point. In each of the original blocks, the sequence is replaced by a `GO TO` *SNn*, where *SNn* is the statement label of the new block. This method, when applicable, is not only highly efficient — even for very short sequences — but also highly elucidates the "commonality" of the code sequence.

ii. However, if the sequences do not all exit to a common point, then a return mechanism must be invoked. An extremely efficient method is the in-line routine, where the return address is defined via the `ASSIGN` statement, and the return is made by an `Assigned GOTO`. The level of the new block which will contain the code sequence should be equal to that of the highest level block originally containing the code. If this level is Level 1 (i.e., a major block), then the new block should be placed ("out of the way") at the end of the module. If this level is less than Level 1, however, then the new block should be located at the end of the higher level block which contains all the sub-blocks in which the code sequence was originally contained. In each of these original blocks, the sequence is replaced by an `ASSIGN` statement, and a `GO TO` *SNn*, where *SNn* is the statement label of the new block. It is very important to note that use of the `ASSIGN/Assigned GOTO` statements in the above-described fashion eliminates *all* objections which have been raised concerning its misuse.

iii. Finally, the third method consists of making the code sequence into a subroutine. Care must be taken, of course, when writing the subroutine to insure that all required variables are input into it. This method insures both high visibility and modularity. Disadvantages are the linkage overhead and possible increase in paging under VS.

[13] The basic intent of "module strength", as discussed in Chapter 3 (Pages 19–31) of Glenford J. Myers' text, *ReliabLe Software through Composite Design* (New York: Manson/Charter Publishers, Inc., 1975), is to organize the totality of elements that constitute an entire program so that "... closely related elements fall into a single module, and unrelated elements fall into separate modules." Interestingly, this concept as applied to all modules which constitute a program, is essentially identical to our concept as applied to the Functional Blocks which constitute an individual module. Thus, once a previously unstructured FORTRAN module is transformed into Functional Blocks, if it is observed that the module strength is low because the module contains a multiplicity of *unrelated* functions, then the points (nodes) at which the module can be separated into two or more modules of higher strength are clearly evident, falling as they do at the boundaries of the appropriate Functional Blocks.

Figure 1: Suggested Eight-Step Iterative Process for Structuring FORTRAN modules

1. Perform a preliminary "clean-up" of the code prior to the actual structuring process by:
   - deleting all *unreferenced* statement numbers,
   - converting those `Arithmetic IF`'s, in which reference is made to the immediately following line of code, into their `Logical IF` counterparts.
2. Determine the series of single function steps required to accomplish the module task.
3. Segment the code into blocks which reflect the series of steps outlined above in Step #2.
4. Functionally renumber the module using the `BLOCKID` renumbering capability of RENUMF.
5. Note any improper `GOTO` branches reported by RENUMF.
6. If no improper `GOTO`'s are found, then go on to Step #7. Otherwise:
   (a) Determine more exactly what is really happening in a block which supposedly performs a single function, but which has an improper branch into its midst. Reconcile this for each major block so affected.
   (b) Go back to Step #4.
7. Now that all improper `GOTO`'s have been eliminated for these Level 1 (major) blocks, then:
   (a) Examine each one-line comment preceding each Functional Block to insure that they are correct (and this implies that the <*event*>'s are unique).
   (b) Determine if process-clarity could be improved by in turn segmenting any of these Level 1 blocks into Level 2 sub-blocks. If so, then Steps #2 to #6 should be repeated. (Of course, once all Level 2 blocks are free of improper `GOTO`'S, you may want to further decompose some of these into Level 3 sub-blocks, again necessitating that Steps #2 to #6 be repeated; etc.
   (c) Look for chains of adjacent blocks to determine if process-clarity could be improved by combining two or more blocks in the chain together.
   (d) Look for sequences of identical code which could be separated out in any one of three different structured ways.
8. Now that the present implementation of the module is clearly structured, then:
   (a) Verify whether the present module algorithm is correct.
   (b) Look for ways in which the present algorithm can be improved.
   (c) Determine whether the module strength can be improved by segmenting the module into two or more distinct modules.