Better manpower utilization using automatic restructuring

by GUY de BALBINE
Caine, Farber and Gordon, Inc.
Pasadena, California

INTRODUCTION

Our intent is to introduce the concept of automatic restructuring as a powerful method for improving the quality of software developed before the advent of structured programming. The quality improvements we are concerned with are neither execution time efficiency nor core size requirements but, rather, higher readability and clear structured code. These, in turn, should improve the reliability and reduce the maintenance costs by making human verification more efficient.

The fact that arbitrary flow diagrams can be mapped into equivalent structured flow diagrams by introducing new Boolean variables has been established by Bohm and Jacopini2 (see Reference 1 for an example of a program that cannot be restructured without additional Boolean variables). The first steps toward systematizing this mapping are taken in Reference 9.

In practice, however, we have found that adding Boolean variables (whose names are meaningless since they would have to be program generated) makes the code often harder to read. Thus Dijkstra's comment4 that “the exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one is not to be recommended” because “the resulting flow diagram cannot be expected to be more transparent than the original one.”

On the other hand, if we allow certain constrained forms of the GO TO statement, many of the difficulties vanish and readability can be enhanced. One form of the constrained GO TO, which we call UNDO is used to exit from nested structures when necessary, the jump always being a forward jump to the end of a DO group. This is similar to the LEAVE statement in BLISS.10

Figure 1 shows an example derived from Reference 1. With the UNDO construct, a natural straightforward representation can be obtained.

Based on the hypothesis that the restructuring process could be applied systematically to existing unstructured programs and enhance their clarity, we have designed and implemented a software tool known as the “structuring engine.” We shall now describe in more detail some of our motivations and the experimental results that we have obtained while using the “structuring engine.”

IMPROVING THE HUMANSOFTWARE INTERFACE

Most production software in existence today was developed using no precise design methodology. The programming languages generally used (FORTRAN, COBOL) were invented over a decade ago and have hardly evolved due to the severe binds imposed by upward compatibility. Maintaining and extending the huge software inventory is a difficult and inefficient task which is becoming even more so year by year. The software documentation is poor, the logic is often obscure, and the authors are most likely to be gone or assigned to other projects. Operational programs still break down with bugs that have managed to escape the most careful scrutiny. Modifications and extensions are dreaded and postponed since they are likely to cause perturbations whose far ranging effects cannot be easily and reliably assessed.

We do not claim to have a panacea that can cure all of these problems instantly. However, the experience gained while developing large scale software using structured programming has shown some of the important factors that influence software reliability and maintenance costs. In our experience, the quality of the human-software interface is one such factor since it influences the efficiency of all manhours invested at the program level, both during development and maintenance.

To benefit from a better human-software interface applicable to future software development, as well as to current software, we suggest extending commonly available programming languages, imposing some constraints to ensure proper language usage, emphasizing the need for visual improvement of programs, and providing transitional tools to assist in the conversion of existing software to meet the new interface specifications.

Language extensions

The only precise, and by definition up-to-date, source of internal documentation for most software in existence today lies in the programs themselves. Understanding what programs accomplish implies an understanding in

319

From the collection of the Computer History Museum (www.computerhistory.org)
the formalism and at the level of detail imposed by the programming language used as a vehicle for implementation. Thus, any shortcomings of the implementation language have a direct impact on the effort needed to understand what the programs do and to modify and extend them successfully.

The two most widespread programming languages, FORTRAN and COBOL, do not contain adequate mechanisms to support structured coding. The limited facilities they provide can be exploited very cleverly to look somewhat like structured code. However, a substantial effort is needed to maintain proper indentation and the legibility is never as good as that obtainable with a structured language.

The obvious step is to build preprocessors to provide the necessary syntactic extensions and perform some of the manual chores such as automatic indentation. Several dozen preprocessors have already been built to translate various brands of structured FORTRAN into pure FORTRAN. Our effort along these lines has led to the design and implementation of the S-FORTRAN language and translator. S-FORTRAN embodies a small but powerful set of structured constructs. S-FORTRAN was designed to serve both as a target language for restructured programs and as an implementation language for new programs. It is not only simple but easy to remember unambiguously. The S-FORTRAN language is succinctly described in the Appendix.

We do not wish at this point to discuss at length the individual merits of each S-FORTRAN feature and whether LOOP is a better term than DO FOREVER or should DO UNTIL test first rather than execute first. These decisions are mostly conventions. Let us simply express the hope that a consensus will soon develop so that a "de facto" standard will prevail. Structured FORTRAN programs will then be unambiguously understood by all.

**Language usage**

Providing extended languages to permit structured coding is not sufficient to guarantee software clarity. Programmers can still misuse structured languages to follow their traditional thought processes, the result being obscure programs under the guise of structured code.

Rather than resort to building enforcement tools, it is our belief that the simplicity and intellectual appeal of a well formed program will generate the necessary motivation among programmers to adopt a new standard of quality.

**Visual improvement**

Structured coding techniques require that programs be systematically indented to stress the relationships between code segments. This hierarchical arrangement allows a quick grasp of the global as well as the local structure of the code. Understanding the code no longer requires keeping track of many scattered items such as labels and transfers. Rather, it means perceiving visual patterns that can be precisely mapped into our analytical understanding of the solution. Each part and subpart corresponds to a block of code, carefully delimited to facilitate its verification. Systematic indentation makes it easy to collect the conditions controlling the execution of each indentation level down to the code segment being examined.

The power of visual perception can be readily tapped by developing patterns whenever feasible. Symmetry, lack of symmetry, block indentation, regularity, recurring patterns, aligned similar items... are characteristics that can be detected at a glance by the eye. Interestingly enough, these are characteristics whose global nature is usually hard to detect and utilize automatically with software tools.

**HOW TO BENEFIT FROM THE NEW INTERFACE**

Formulating a better human--software interface is clearly valuable for software that has not yet been written. The important point is that such an improvement can also be applied to a large part of the software inventory in existence today. It is our belief that this should lead to a significant reduction in the maintenance effort by better utilizing the available manpower.

Until very recently, the main route for modernizing existing unstructured software was to start over with a clean top-down design and structured implementation. Needless to say, such complete manual reprogramming should not be undertaken without a very careful evaluation of the potential gain versus the effort involved. We have found that the major obstacles to manual reprogramming are the need for top talent during the redesign phase, the manpower expenditure, the elapsed time before a new
Figure 2—Sample program FORIT before and after restructuring
As an alternative, we have developed a method which is much easier to apply in practice. It consists in keeping the global design as is, in particular the data structures, and in automatically transforming every program into an equivalent structured program, visually improved to make its reading easier and its understanding more thorough. This method is supported by a software tool known as the “structuring engine.” We have applied this tool to a variety of FORTRAN programs. We shall now describe the characteristics of the tool and some of the experimental results that we have obtained so far.

THE "STRUCTURING ENGINE"

Capabilities

The “structuring engine,” as it now exists, is a large task running on an IBM/370 under VS. It consists of over 30,000 lines of structured PL/1 code. It will restructure programs written in FORTRAN including any language extensions acceptable by IBM, Univac, CDC and Honeywell compilers.

Each program or subprogram is restructured independently. The complete flow graph of each program or subprogram is analyzed to determine the best strategy for obtaining a well structured program. Machine dependencies are taken into account when building the flow graph because the interpretation of some statements depends on the particular compiler that the program was intended for. For instance, values outside the range of a computed GO TO can be handled in three distinct ways depending upon the particular compiler implementation. Such variations are taken into account by the "structuring engine" which generates the necessary statements to guarantee consistency in the restructured output.

In general, the restructured programs will bear little resemblance to the original unstructured ones, particularly if the logic was complex and somewhat twisted to start with. In the output, the logic flows from top to bottom, from the single entry to the single exit.

Figure 2 is an example of a simple program before and after restructuring. Similarly, Figure 3 shows what happens in the case of a heavily folded program. The restructured programs are equivalent to those from which they are derived in the sense that they behave identically at run time. That is, they carry out the same sequence of operations on the data structures, great care being taken that the ordering of operations not be modified. For instance, a three way arithmetic IF cannot be simply converted into two nested S-FORTRAN IF statements because the arithmetic expression would then be evaluated twice. In that case, incorrect results might be obtained if the arithmetic expression contains calls to abnormal functions i.e., functions which do not always produce the same results from a given set of inputs.

One of the basic processes used in restructuring is known as node splitting. If a node of the subgraph can be reached from two different paths that must be separated, the node is split into two identical nodes so that each path can have its own copy of the node.

If the node splitting operations were carried out indiscriminately, the resulting S-FORTRAN programs would often become so large as to be virtually useless. Not only would clarity be lost but the object program would be likely not to fit in the target machine. To circumvent that
difficulty, the “structuring engine” tries recognizing proper subgraphs that can be turned into procedures instead of being duplicated in line. A procedure is simply a section of code with one entry and one exit. This concept corresponds to the PERFORMed group in COBOL, but with additional constraints to guarantee a clean invocation and a clean return. Once a procedure has been extracted and given a name, it can be referencing many locations within the restructured program, including from other procedures. The example in Figure 2 contains one procedure, the one in Figure 3 contains two. The decision whether to expand code in line or create procedures can be externally controlled using a threshold which indicates how complex a subgraph must be before it becomes a procedure. Procedures are not separate subprograms. Rather, they are segments of code that can be executed from various locations within a particular program or subprogram. The EXECUTE command hides the ASSIGNed GO TO linkage that a FORTRAN programmer would have to set up otherwise.

To visually improve the resulting code, every statement is laid out according to its logical indentation level. This stresses its relationship with other statements in the same program unit. A box is built around each complete DO group to enhance the scope of the DO statements. Statements such as UNDO, CYCLE, and RETURN are followed by an arrow that attracts the attention of the reader and shows him immediately what implications the statements have on the logic flow. Consecutive comment cards are right adjusted by block in order to make them as unobtrusive as possible.

Of course, if the modules to be restructured contain logic errors, the same errors will be found in the structured output. In general, the “structuring engine” is incapable of detecting errors except for some obvious language violations. The input programs are supposed to have compiled correctly so that we are really trying to eliminate are errors in the logic that cannot be identified without an intimate understanding of the problem. Only a programmer aware of the problem being solved can discover and correct these errors.

Emphasis in building the “structuring engine” has been on reliability rather than efficiency. This has been achieved through a combination of structured design techniques, self identifiable data structures and dynamic assertion verification at run time i.e., the constant verification that the assumptions underlying the design are never violated during production runs.

Experimental results

We are currently applying the “structuring engine” to a wide variety of unstructured FORTRAN code. Although our analysis is far from complete, we would like to comment on some of the experimental results that we have obtained so far.

![Figure 3—Sample program ORDB unstructured (Part I)](image)

Clarity of the restructured programs

A reliable assessment of clarity improvements is obviously quite difficult to obtain until we get some figures on maintenance costs. The familiarity of the end user with structured code is a factor as noted in Reference 6. The cleverness of the “structuring engine” in making the right choices is obviously another important factor since there are not one but many solutions to the restructuring problem. So far, we have found that:

- the majority of the programs (about 90 percent) will come out extremely clear, at least in our opinion and in that of end users that have worked with restructured programs.
- the rest (about 10 percent) will either remain complex or become lengthy or both. In this group, we find a number of programs that could be handled more cleverly by the “structuring engine” and, therefore, move into the above category. We are obviously building the necessary improvements into the “structuring engine.” Still, there are some programs that will probably never look very good. They are ill-designed. The problem that they are supposed to solve should be reexamined and a complete redesign and reprogramming of these programs may be necessary.
Figure 3—Sample program ORDB restructured (Parts II and III)
Execution characteristics of the restructured programs

Let us now try to answer some of the most common questions regarding this automatic restructuring process. What price do we pay for the improved clarity of the restructured programs? In particular, how do the restructured programs execute compared to the original ones?

To answer these questions, we must first examine the various components in the processing chain as shown in Figure 4. The "structuring engine" transforms unstructured FORTRAN into structured S-FORTRAN. The resulting S-FORTRAN programs are then translated back into FORTRAN using the S-FORTRAN to FORTRAN translator. At that point, we have pure FORTRAN source code again which can be compiled, loaded and executed. Thus, the characteristics that we are reporting on involve not only the "structuring engine" but also the translator and a compiler.

The core size of the object modules produced from the restructured programs has been found to always be larger than that of the original modules, typically by about 20 percent. We know from Reference 9 that arbitrary programs cannot be restructured without increasing their running time or their core size. In the present case, we have chosen to accept a limited increase in memory size. The creation of internal procedures is our method for preventing a program from growing beyond an acceptable point.

Figure 5 shows a typical distribution of core size expansion ratios (1 would mean no increase) as a function of the size of the object module for the unstructured program when compiled with IBM's FORTRAN G compiler. The circled data point corresponds to a program that the "structuring engine" could not structure without producing three times as many S-FORTRAN cards as there were FORTRAN cards. This "abnormal" expansion factor was caused by a deeply nested section of code that could not be turned into procedure because it would then have contained an UNDO outside its scope. Such an UNDO outside the scope of a procedure is not permitted in S-FORTRAN. Consequently, the same section of code was duplicated 18 times throughout the program.

Data on the execution speed of the restructured programs has been harder to get because most of the programs we have restructured so far were components of much larger systems which we could not run ourselves. Preliminary results show that we should expect slight variations in the running time with a trend toward a re-

<table>
<thead>
<tr>
<th>OBJECT MODULES</th>
<th>STRUCTURING ENGINE</th>
<th>ONE TIME TRANSLATION</th>
<th>S-FORTRAN TRANSLATOR</th>
<th>S-FORTRAN TO FORTRAN COMPLIER</th>
<th>FORTRAN COMPLIER</th>
<th>MAINTENANCE AND/OR DEVELOPMENT OF RESTRUCTURED PROGRAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN UNSTRUCTURED</td>
<td>STRUCTURED FORTRAN</td>
<td></td>
<td>S-FORTRAN STRUCTURED</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 4—The restructuring chain](From the collection of the Computer History Museum (www.computerhistory.org))
duction rather than an increase. This may seem para-
doxical at first but can be explained as follows. There are
two major factors that influence the running time in op-
posite ways:

(i) the size of the basic block: the restructuring process
cannot decrease the average size of the basic block
and in general will increase it. Thus, an optimizing
compiler should generate better code within each
basic block of the restructured programs.
(ii) the control flow statements produced by the transla-
tor to support branching, looping and procedure
referencing: These require, in general, more
instructions than needed to implement the original
control logic.

The first factor tends to make restructured programs
run faster whereas the latter tends to slow them down.
This means that with a translator generating very good
code we should be able to have programs run faster
restructured than unstructured. In fact, we have now built
more sophistication into the translator than had been
originally planned in order to make full use of the capabil-
ities of optimizing compilers. For instance, with IBM's
FORTRAN H (OPT=2) compiler, changes in the transla-
tion of IF statements have reduced in one instance the
core size by 12 percent and the running time by 8 percent
when compared with earlier versions of the translator.

CONCLUSIONS

Automatic restructuring as implemented by the "structur-
ing engine" is proposed as a method to modernize existing
programs. It should prove much more practical than
manual reprogramming, particularly with regards to man-
power requirements, conversion time and the reliability of
the conversion process itself.

Manpower requirements are reduced since no major
human effort is invested redoing what already exists. On
the contrary, programmer time is devoted to perusing
restructured programs, implementing improvements
wherever deficiencies show up, and correcting errors. In
particular, any program which still appears to be overly
complex after restructuring compared to what it is sup-
posed to accomplish, becomes a good candidate for an in-
depth investigation of the reasons underlying its apparent
complexity. Poor algorithms may be pinned down fast and
replaced accordingly. The overall result is that the
programmer understands the structured code more
rapidly and can, therefore, allocate more time to difficult
areas. Consequently, his error detection rate increases,
thus justifying our claim to improved software reliability.

Conversion time is negligible compared to that required
for manual reprogramming. In particular, the project's
clock is not set back since the restructuring process does
not introduce any new errors.

Of course, there may still be cases where complete
redesign and reprogramming appear to be absolutely
necessary. Under those circumstances, the "structuring
engine" can still play an important role. Indeed, no matter
how unstructured and clumsy the original software may
be, it represents an approximate solution to the problem,
correct in most if not all of the cases. As such, it acts as a
repository for a wealth of details that were added
throughout the life cycle of the software to handle unusual
and certainly unforeseen cases. Starting from this rich
data, the "structuring engine" becomes a very valuable
tool since it produces an up-to-date structured picture of
the solution currently implemented. This picture may
then be used to base a thorough evaluation of the status of
the project, including any needs for manual redesign and
reprogramming.

APPENDIX

The main characteristics of S-FORTRAN are:

(a) S-FORTRAN is a superset of FORTRAN (including
the FORTRAN language extensions provided by
various manufacturers).
(b) Any construct with a scope has both an opening and a
closing delimiter. If the opening statement is XXX,
the ending statement is of the form END XXX. (e.g.,
IF . . . END IF, DO WHILE . . . END DO WHILE).
(c) The IF includes any number of ELSEIF clauses and
an optional ELSE clause. ELSEIF's are often
convenient to prevent very deep indentation levels (and
the so-called "wall to wall" ENDIF's).
(d) Repetitive DO groups include a DO FOR analogous to
the FORTRAN DO loop, a DO WHILE, a DO
UNTIL (which is in fact a DO AT LEAST ONCE
UNTIL), and a DO FOREVER (an infinite loop).
(e) Non repetitive DO groups include a DO for bracketing
statements, a powerful DO CASE, a DO CASE SIGN
OF which is the equivalent of a three way arithmetic
IF, and a DO LABEL to handle abnormal returns
from subroutines and functions and end and error
exits from I/O statements.
(f) UNDO is a mechanism to exit from a DO group prematu-
ately. We have found this multilevel exit
mechanism to be superior to introducing switch
variables which tend to clutter the program and make
its logic harder to follow. UNDO is applicable to any
DO group, repetitive or not. It can be followed by a
label if another DO group besides the innermost one is
to be exited from.
(g) CYCLE is similar to UNDO but implies skipping any
statement until the closing delimiter of a DO group is
found. The test controlling the repeated execution of
the DO group is then performed to determine whether
to exit or repeat. CYCLE is only applicable to repeti-
tive-DO groups.
(h) Internal parameterless procedures can be defined us-
ing PROCEDURE . . . END PROCEDURE. Their
execution can only be triggered by an EXECUTE
(proc-name) statement. Premature termination of a procedure can be accomplished by an EXIT statement. Procedures share the same data space as the program in which they are contained.

REFERENCES
