Version of article with the same name published in IEEE: Computer, 1994.

Converting Legacy Code into Ada: A Cognitive Approach

Dr. Joseph M. Scandura, University of Pennsylvania; SCANDURA-Flexsoft International LLC

Introduction

This paper reviews the current status of the re-engineering industry and then proposes a new cognitive approach to system maintenance which can both dramatically improve systems and minimize Ada renewal costs. This cognitive approach involves modeling and testing the structural and functional essence of a system at a high level of abstraction, with increasing specificity until contact is made with available data and computational resources. The process is essentially the same whether structural analysis (i.e., the cognitive technology) is used to design and develop new systems or to re-engineer old ones. In the former case, the to-be-developed system exists only in the mind of the analyst, designer and/or end user. In the latter case, one begins with a fully functioning system. In both cases, heavy use is made of reusable routines (with new systems) and/or of code salvaged as a result of re-engineering. Conversion to Ada pertains largely to how a system is physically represented.

A CASE history and an example round out the discussion, the latter in the context of Phase II SBIR research for the U.S. Army. The results demonstrate that high quality results can be achieved in large part automatically, and at greatly reduced cost.

Saddled with large quantities of obsolete but essential COBOL or FORTRAN, Department of Defense contractors and agencies are faced with a series of unpalatable choices. One option is simply to continue with the same old software, patching it where possible to meet the most pressing needs. Properly implemented, reengineering can add a measure of efficiency previously unattainable. Although costs for renewal can often be recovered over time, initial costs are often prohibitive, however. Decision making is further complicated by DOD directives to convert systems requiring significant change (about 30 percent) entirely to Ada.

Faced with this dilemma, what are decision-makers to do? Indeed, reengineering (including reuse and maintenance) is today one of the "hottest" topics in software engineering. There is a good deal of confusion, however, as to just what reengineering involves, and even more concerning the benefits of current reengineering tools.

In this article, I first review current software reengineering tools and then describe a new cognitive approach to system (re)engineering based on code comprehension tools that provide a visual representation of code containing less "cognitive noise." This better enables programmers to understand system design. Our approach integrates code comprehension tools with current reengineering methodologies to create an integrated reengineering workbench for converting legacy code into newer languages such as Ada or C/C++.

Current Approaches To Software Reengineering

Several classes of reengineering tools have evolved over the past few years, offering software engineers an array of choices. These tools have both strengths and weaknesses, and their effectiveness hinges on effective representation of the visual and physical aspects of the system. Successful conversion to Ada, for example, depends largely on how a system is physically represented.

Code analysis tools. Code analysis tools help gather useful information about existing systems, such as complexity measures, calling hierarchies, cross-reference lists and information about the organization of code (or the lack thereof).

However, tools in this category have a major limitation. They provide information about a system but do not support making needed changes to the code. Thus, programmers can gain insights from the analyses, but they still have to find the source of the problems and ways to fix them.

Restructuring tools. Certain kinds of code modifications can be automated. Commercially available tools exist, for example, to automatically restructure FORTRAN and COBOL source code. With FORTRAN, restructuring is almost always desirable. Eliminating GOTOs significantly facilitates code understanding from a cognitive point of view. Instead of having to scan and integrate scattered code fragments, software engineers have immediately relevant processes available in one context. COBOL restructuring poses a unique set of problems, however, because GoTos cross module (paragraph) boundaries. Eliminating GoTos and "fall-throughs" generally results in what amounts to entirely new COBOL programs (Perry, 1989). Nonetheless, if the goal is long-term maintainability, COBOL restructuring is an essential first step.

Design recapture tools. A second, somewhat newer class of tools is concerned with design recapture -- analyzing source code to determine and visually represent relationships between source code modules. Typically, the information obtained is represented in some type of structure chart or module hierarchy.

The basic technology generally involves simple parsing techniques in which modules are identified and attendant relationships are captured for later visual representation. The process is not unlike extracting a table of contents from a book. Extracting overall relationships within a system and representing them in a visual environment (where they can more easily be modified) is clearly worth doing. Unfortunately, one cannot rely on overall structure in making changes to code. As any programmer knows, the "devil hides in the details."

Most reengineering tools provide limited access to module code. Some can access editing tools with which to modify such code. This approach still leaves the biggest problem -- understanding details in order to know how to modify the actual code. Traditional GUI (graphical user interface) representations simply *do not* lend themselves well to this task.

Module visualization. Source code is not easy for the neophyte to understand. Cognitive studies show that understanding code can require considerable effort even for skilled programmers. Programmers spend most of their time (about 90 percent) understanding code and only 10 percent making changes. Consequently, anything that can facilitate understanding will pay handsome dividends.

"Pretty printing," though a step in this direction, is not sufficient. Action diagrams (Martin and McClure, 1988) help further by organizing code structure; they bracket code, making it easier to perceive structure groupings. It remains for the programmer, however, to distinguish different types of structures and to separate relevant code from irrelevant detail. Action diagrams, for example, still contain extraneous syntax and bracketing information (such as,],},;, Begin, End). Studies show that irrelevant information, even when highly familiar, increases what is called "cognitive strain." The higher the cognitive strain, the lower the capacity for productive thought.

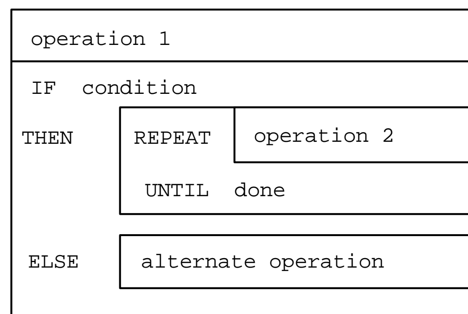

Figure 1a: Flexform representation with standard borders.

```
oaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa#
b operation 1    b
baaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
b IF condition   b
b                oaaaaaaaaaaaaaaaaaaaaaaaa
b THEN           b REPEAT    b  operation 2   b
b                b        caaaaaaaaaaaaaaab
b                b UNTIL     done             b
b                   caaaaaaaaaaaaaaaaaaaab
b                   oaaaaaaaaaaaaaaaaaaab
b ELSE           b alternative operation     b
b                caaaaaaaaaaaaaaaaaaaaaaaax
Laaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaax
```

Figure 1b: Identical contents bracketed with "alphabetic" borders.

The visual bracketing that is provided for different structures in FlexformS can improve comprehension. Flexforms are based on a unique contextual display process (U.S. Patent 5,262, 761) in which each type of procedural refinement (e.g., sequence, selection. loop) has a perceptually distinct form. Flexform's are highly dynamic and flexible in appearance making it possible to represent procedural logic at any and/or levels of abstraction, from the highest to the lowest levels. To get some appreciation for the perceptual advantages of Flexforms, contrast Figures 1a and 1b. The surrounding lies in Figure 1a are directly and automatically perceived as distinct from the contents, whereas the bracketing letters in Figure 1b are of the same genre as the code.
If these diagrams had been compressed vertically, the perceptual difference would have been even greater. Module visualization aids human comprehension by representing structure visually and eliminating irrelevant detail.

Although better visual representation helps, this alone is not sufficient. To improve on code analysis tools, it must be possible to automatically construct such visualizations from code and to modify the code directly in the visual environment. In this context, an integrated reengineering workbench, should automatically reverse engineer existing code into pseudocode Flexforms. The tool should also provide an interactive environment where Flexforms can be edited, documented, restructured and customized to support multiple environments and should regenerate full source code as desired.

Contextual versus separate windowing. Two kinds of representation are implicitly described above: representation of relationships (e.g., structure charts) and representation of modules. Rectangles and circles connected by lines (bubble charts) are commonly used for these purposes. They are inadequate for other purposes, however, especially real-time systems, and they are hardly unique. Flexforms, for example, can be used to represent dataflow diagrams, structure charts, control-flow and entity-relationship diagrams and context diagrams (Scandura, 1992).

It is, nonetheless, important to distinguish different kinds of information to be represented. Each kind of information deals with a very different aspect of a system: modules, module relationships and file/unit relationships, for example. Different kinds of representation are best displayed and edited in separate windows. Structure charts, for example, convey information about relationships between modules. They are intrinsically different from the modules themselves. The same thing can be said about modules or about relationships between compilation units or files.

Separate windows are *not* desirable, however, when talking about different levels of abstraction within the same type of representation. Displaying such information in different windows places severe restrictions on the number of levels that can be displayed (on a screen) and their ease of comprehension. Expansion in different windows, for example, makes it difficult to remember which windows (expansions) go with which elements in other windows. Most experts agree that it is difficult to understand more than three or four levels of a dataflow diagram at one time.

One solution to this problem is to use some form of contextual windowing. Flexforms accomplish this by allowing "explosion" directly in context. Lower level detail is automatically displayed within the element that contains it. Thus, Flexform rectangles can be expanded without affecting the context above or below. This lets us see more detail without losing the general picture. (It is well known in cognitive psychology that the number of different "chunks" of information that a human can deal with simultaneously approximates the "magic number 7 plus or minus 2" [Miller, 1956; Voorhies and Scandura, 1977].)

Graphic elements, such as boxes or circles, connected by lines lack this feature. Expanding an element in a bubble chart simple changes the overall scale. Consequently, the original context disappears off the monitor screen (Scandura, 1992). In contrast, Flexforms let us view module relationships, unit relationships and even program relationships at any desired level of abstraction (Figure 2).

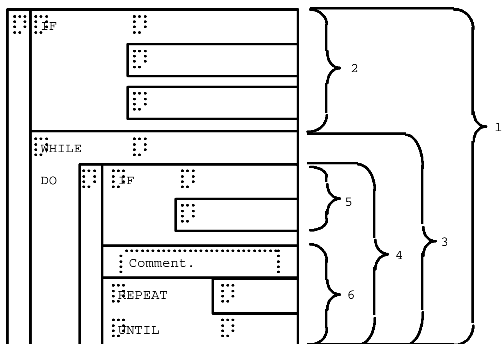

Figure 2. This diagram is reproduced from US Patent No: 5,262,761 by Scandura et al. (Nov. 16, 1993). It shows several levels of a Flexform hierarchy with cursor positions corresponding to different levels of the defined tree-like structure. The number 1 corresponds to the top-level structure, 2 and 3 correspond to 1's child structures, 4 to the body element (child) of 3, 5 and 6 to children of 4. All except 6 show terminal elements only. "Comment" within 6 refers to the tree element immediately above the terminal elements of 6. Higher level elements are displayed inside a distinguishable border (a dotted frame). "Fanning in" and "fanning out" in tree-like structures is accomplished via visually distinguished clones, but none is shown in the diagram.

Of course, different kinds of representations (for example, modules or call hierarchies) are represented in different windows. Flexforms differ from other views in that the same intuitive representation is used throughout. Moreover, the user never has to look at code per se, even at the very lowest levels of module detail. All desired modifications should be made directly in Flexforms without restriction. Any program that can be written in Ada, for example, can be written directly in Ada Flexforms.

Conversion to new environments. Given continuing improvements in hardware and operating systems, we are often faced with the task of converting old software to new environments. To make matters worse, we frequently need to maintain two or more versions simultaneously. Normally, this is accomplished by separate teams of programmers maintaining two or more sets of files. Obviously, it would be better if we could maintain multiple versions in one set of files.

One solution is to use conditional compilation metacommands supported by most compilers. This approach, however, clutters the code and makes it increasingly difficult to read and understand program logic. Metacommands provide extra "noise" which, as cognitive studies show, negatively impacts human comprehension.

The amount of irrelevant information is minimized in Flexforms by labeling structures that are unique to a given platform or operating system. These labels are used during code and/or report generation to automatically produce multiple versions for different environments on demand. Consequently, only the Flexforms need to be preserved.

Conversion between languages. In moving to a new environment (for example, from IBM's MVS to PC-DOS or UNIX), it is often desirable to convert from an existing programming language into a more modern one, such as C/C++ or Ada.

Some have argued that the only reasonable way to accomplish this is to rewrite the code. One argument against this approach is cost. A less obvious limitation is that the code produced by the level of programmer likely to be assigned to the task might not be much better than the original.

A second approach involves source-to-source translators that take source code in one language and convert it directly into source code in another. A common complaint about such tools is that they result in poor Ada -- "AdaTRAN" or "AdaBOL," for example.

A third approach involves converting the source language into an intermediate form having semantic as well as syntactic characteristics. Code in the target language is generated from this intermediate language. Better results can often be achieved in this manner because the approach generally makes it easier to deal with semantic as well as syntactic issues. The overall process, however, tends to be slower and more complex. High-level reengineering issues also tend to go unaddressed.

A fourth approach refines the intermediate language technique by more sharply distinguishing syntactic and semantic aspects of a translation. Efficient parsing technologies are used to rapidly complete syntactic aspects of the conversion, leaving semantic issues for more powerful and normally slower semantic transformations. In general, syntactic transformations work best up to the individual statement level. They also can be adapted to map multiple statements (for example, FORTRAN Format and Print statements) into the target language. Semantic transformations become increasingly necessary as the mappings address deeper (or more abstract) semantic differences (for example, FORTRAN Commons versus Ada packages).

What most differentiates this fourth approach is that the conversion works both syntactically from the bottom up and semantically from the top down. The term "cognitive" aptly describes this approach. Bottom-up analysis corresponds to the largely automatic processes a skilled programmer uses in line-by-line conversions. Top-down analysis corresponds to the more thoughtful analysis that goes into making high-level design decisions. The results of such conversions can approach or even exceed those performed by an average human programmer. There are, of course, practical limits to what any generic translation tool can do. Languages, language definitions and compilers come in many variations, not to mention differences in operating systems and libraries. Consequently, a full solution to the conversion dilemma must lend itself to customization.

Automatic conversion can be accomplished in two steps. The first involves reverse engineering the source code (for example, FORTRAN, COBOL, or C) into a modular, object-oriented Flexform repository. (This modularity is ideally suited for client-server environments where information may be scattered.) Once reverse engineered, the semantics as well as the syntax of the source code are directly accessible. (Reverse engineering normally is fully automatic, although minor customization may be necessary with nonstandard code.) The second step involves conversion. Visual Flexforms containing pseudocode in one language are converted into Flexforms containing pseudocode in another language. Parsing techniques perform the simpler syntactic conversions. In turn, Ada semantic postprocessors can take the translator output and turn it into good Ada. "Good" in this context means that Ada constructs (such as packages) are used that have no direct counterpart in the source language. It does not

necessarily imply that the results would be indistinguishable from that produced by an Ada expert.

Both the syntactic and semantic post-processing aspects are customizable. The basic machinery also is extensible to new languages. Currently, C/C++ and Ada conversions from FORTRAN, C, Pascal and COBOL are supported. From 90 to 99 percent of the code is converted automatically, with higher level designs preserved in the process. These levels of automation may be further improved through customization. Customization is straight forward once it has been decided exactly what is to be done and under what circumstances.

Our experience suggests that a minimum of 50 percent of existing code -- and usually much more -- is reusable.

To summarize, source-to-source and intermediate-language translators represent a reasonable approach if no further maintenance on the code is desired. But if this were the case, why translate the code to begin with? In general, we translate code because the software can be better maintained in the new language. It is widely recognized in the DOD, for example, that Ada programs are easier to maintain than programs written in FORTRAN, COBOL, C, or Jovial.

System redesign. The desire for continuing enhancement implies a need for conversion capabilities that make explicit provision for reengineering. In short, many situations call for creating entirely new or renewed designs. Rather than building an entirely new system, however, it is possible in most cases to salvage much reverse engineered or converted code. Reusable code can be either highly specific or relatively comprehensive. In most cases, it should be relatively modular. Reusing code from an existing system to build a better system in the same domain has the major advantage that large, high-level modules can often be reused in implementation. Our experience suggests that, at a minimum, 50 to 60 percent of existing code -- and usually much more, sometimes as much as 99 percent -- is reusable in redesigned systems. The key to reusability is not simply the quality of code. Code stability over time can be even more important. As long as the black box works and is not likely to change, there is little need to “look inside.”

Most front-end CASE tools support new design. Some also support simulating display and input screens, largely to ensure user satisfaction. Both of these factors (that is, design and interactive display of user screens) play an important role in system design or redesign. But they are not the only factors. Confidence in a new design comes only from testing (and debugging) underlying logic.

Testing is expensive and time consuming, even with the assistance of test generation tools. The standard approach involves both unit and integration testing -- a long, often arduous process. This approach poses a fundamental problem in that it is impossible to test all paths, even equivalence classes of paths (Scandura, 1973). The number of tests required increases exponentially if all testing is done after implementation (Scandura, 1990). Conversely, the number of tests only increases additively if testing is done from the top down. Approximately 2^{100} empirical tests are required in the example cited (Scandura, 1992) when one waits until complete implementation before testing. Only about 300 tests are required when testing is done successively from the highest levels of abstraction.

Testing at the design level requires some form of executable specifications, which unfortunately often require learning an entirely new language -- a hindrance that can greatly reduce overall benefits. Formal specification languages are required largely because the commonly used design methodologies favor either data analysis or process analysis. Lacking a balanced approach to data and process, they do not lend themselves to debugging designs.

A cognitive approach to systems design demands that data and process be considered in parallel. Arbitrarily abstract specifications may be used with respect to both data and process. “Destroy (missile),” for example, is as adequate a specification from a cognitive point of view as “add (A, B).” Comprehension in the former case simply requires a more sophisticated human interpreter. The essential requirement for design-level testing is that data and process both be represented at the same level of abstraction.

Interfacing renewed designs. Creating and testing a high-level design is only one part of the problem. The high-level design must be interfaced with reverse engineered or otherwise reusable code. The current solution is to automatically convert high-level designs to the target language and to create an interface between converted designs and the reusable code. Checking processes in turn provide an interactive, semiautomatic way to create links between converted designs and the data/process resources referenced in those designs. Source code generated from pseudocode Flexforms can be compiled and linked directly to the reusable code.

Automating system redesign. Another way to improve a software system is to manipulate system semantics under program control. Flexforms facilitate this process because they directly expose program semantics thereby facilitating such manipulations. For example, in any given application, the database is typically accessed in a relatively small number of different ways. Each access method corresponds to a semantics-based pattern which (e.g., with an appropriate set of semantic-based routines) can easily be detected and automatically modified (e.g., to support a new database).

Cognitive Approach To System Renewal

Implicit in the above discussion is an integrated cognitive approach to system development, reengineering and conversion, which shares certain attributes with Boehm’s spiral model and Yeh’s programming by design (Boehm, 1988; Yeh, 1990). This approach involves modeling and testing a system’s structural and functional essence at a high level of abstraction, with increasing specificity until contact is made with available data and computational resources. An essential characteristic is that both data and process must be represented at the same level of abstraction. This is analogous to the representation of human knowledge (Scandura, 1973; Scandura, Durnin and Wulfeck, 1974). The more knowledgeable the human population being modeled, the more abstractly the knowledge in question can be represented. Put differently, the larger the “chunks” or atomic rules, the more easily any system (or body of knowledge) can be modeled (Miller, 1956; Voorhies and Scandura, 1977). Testing and diagnosis of individual knowledge at higher levels of

abstraction is more efficient because fewer paths are involved (Scandura, 1973).

First used to construct representations of human knowledge, structural analysis has equal applicability in software engineering.

Testing and debugging software designs makes sense only when data and process are represented at the same level of abstraction. “Structured analysis” with its emphasis on process is inadequate because it neglects the importance of data. Conversely, deferring the representation of processes in “information engineering” leaves no actions to test. Object orientation has limitations as regards testing and operation abstraction because objects are not the same as operations on objects (Scandura, 1994). The former correspond to automatic human perception, the later to conscious cognition.

It will suffice here to call attention to the method of “structur*al* analysis” (the “*al*” is intended [Scandura, Durnin and Wulfeck, 1974; Scandura, 1984]). First used to construct representations of human knowledge, structur*al* analysis appears to have equal applicability in software engineering. The process is essentially the same whether structural analysis (that is, the cognitive technology) is used to design and develop new systems or to reengineer old ones. In the former case, the to-be-developed system exists only in the mind of the analyst, designer and/or end user. In the latter case, we begin with a fully functioning system. In both cases, heavy use is made of reusable routines (with new systems) and/or of code salvaged as a result of reengineering. This is true whenever code is salvaged from a legacy system, whether as is or after conversion to Ada or C++.

System renewal involves modeling the behavior of a designed system from the highest levels of abstraction. In parallel, existing code is reverse engineered into a modular repository compatible with that model. The language-independent model and the reverse-engineered code may optionally be converted into the same target language (for example, C/C++ or Ada). Finally, the debugged model is linked to reusable modules in the legacy code. The reused code is supplemented as necessary with other libraries and/or new modules (Scandura, 1992; Scandura, 1990; Scandura, 1994).

A short case history. A short case history illustrates the overall process. Motivated by the results of 20 years of basic research in the cognitive and computer sciences, PRODOC (as FlexSys was known in those days) was begun as a self-funded research project in 1983 when I was a professor at the University of Pennsylvania. The original goal of the research, which is only now being realized in FlexSys, was to automate software development and maintenance processes based on principles derived from the Structural Learning Theory (e.g., Scandura, 1971, 1973, 1974, 1977). PRODOC initially was designed to provide the necessary infrastructure. By early 1984, it was increasingly used in its own development. Since 1985, all further development and enhancement has been done exclusively within PRODOC itself. By mid-1986, even the early "bootstrapping" routines had been reverse engineered into PRODOC.

Therein lies the genesis of the Cognitive Approach re-engineering. There were almost 100,000 lines of bootstrapping code, about half of which related to PRODOC's dynamic simulation and prototyping facilities. This code was functioning correctly and it was doing essentially what was wanted at the time. Like most production code, however, the code had gone through more than its share of revision. Patch after patch had left that part of the system extremely fragile. It was a major chore to make even the most trivial changes for fear of introducing unanticipated interactions. Introducing major enhancements on our "wish list" was unthinkable.

Even though the simulation and reverse engineering facilities were not nearly as advanced as they are today, they were sufficient to support essentials of the cognitive approach. First, the original code was reverse engineered. The results were dramatic once pseudocode was extracted and "uploaded" into Flexforms.

The structure of the pseudocode still mirrored the twisted design. Still, the visual representation of that design made the overall organization clearer and what had been unthinkable became at least possibility.

The lower level routines generally were not in bad shape. In fact, many were quite good. Adding higher level annotation in those cases improved the situation even more.

The current functioning of the system, however, and the current organization of the code were largely out of sync. To better understand the current system, we used Flexforms to model the then current functionality of the system (with an eye toward planned extensions). The design was tested dynamically for verification purposes.

It was at this point that we decided to gamble. Rather than try to salvage the original code, we took the new design as the starting point and mapped the reusable lower level routines into it. Aside from renaming a few variables and the like, we were surprised (actually amazed) that the entire "mapping" process took only two days plus another half day of debugging. It is important to emphasize, however, that we had been careful in planning and testing the new design. And, in fully understanding the reused lower level modules.

Application of the methodology and maturation of the toolset. These methodologies and supporting tools have matured significantly from their inceptions in the early 1980’s and have been applied in a wide variety of situations. The most recent applications were part of a recently concluded Phase II SBIR project on “improving Reliability, Availability and Maintainability (RAM) of Large Software Systems” supported by the U.S. Army at Picatinny Arsenal under the technical direction of Dr. Mort Hyman. (A two page abstract of the final report and a full copy of the final workshop transcript are available at http://www.scandura.com.) One of these projects involved converting from FORTRAN to Ada. An even larger commercial project (which also was integral to the research) is described in the case history below.

The FORTRAN to Ada conversion process is shown in Figure 3 on the next page. Figure 3a shows sample FORTRAN code, and Figure 3b shows the FORTRAN after restructuring and reverse engineering into a Flexform. The structure of the procedure, including control flow, is immediately apparent without special training. As noted earlier, further benefits derive from the ability to collapse or expand Flexform structures directly in context.

```
C     SUBROUTINE MYSUB
      …
      IF (D.LT.A) GO TO 25
      B = Y
      RETURN
25    CONTINUE
      IF (D.LT.A-B) GO TO 45
      IF (B.GT.Z) GO TO 35
      B = Y
      RETURN
35    B = W
      RETURN
45    B = Z
      RETURN
      END
```

Figure 3a: Simple FORTRAN module.

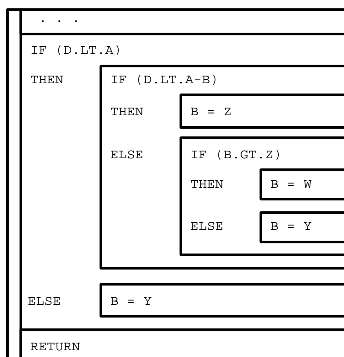

Figure 3b: Corresponding FORTRAN pseudocode Flexform after reverse engineering and restructuring to remove GOTOs.

An example of passing the FORTRAN Flexform (Figure 3b) through the PRODOC Translator is shown in Figure 3c. In this case the FORTRAN code is so simple that the translator does all of the work directly. The translator also handles more complex procedural constructions. FORTRAN Print statements, for example, are keyed to associated but possibly distant Format statements. Both are needed to determine the corresponding Ada output statements.

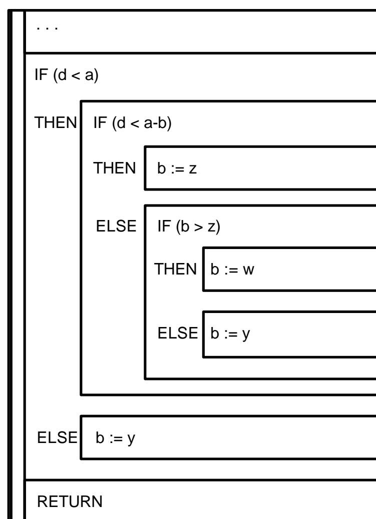

Figure 3c: Ada pseudocode Flexform obtained by translation. From here, full Ada source is immediately available.

In most cases, semantic postprocessing also is a necessity. FORTRAN arrays, for example, are initialized in Data statements, which are separate from variable and type declarations. These are defined in Ada as arrays, but with initializations in procedural sections. Figure 4 shows sample FORTRAN source and the corresponding Ada Flexform and source code.

```
PROGRAM TEST
REAL SIDE2(8)
REAL PI
LOGICAL DONE
INTEGER SIDE
CHARACTER*4 NAME(2,2,2)
DIMENSION SIDE(5)
DATA NAME/'A','B','C','D','E','F','G','H'/
DATA SIDE2/8*-0.5/
DATA DONE/.TRUE./
PARAMETER (PI=3.141593
C Procedure section of sample FORTRAN program
END
```

Figure 4 (a) Sample FORTRAN with a variety of data constructs

```
[examplef.fl2]:TEST;[]ada;[procedure];

  [VARIABLE]:

    [done]{BOOLEAN}:TRUE

    [name]{array (1..2, 1..2, 1..2) of name_STRING}:

    [side]{array (1..5) of INTEGER}:

    [side2]{array (1..8) of FLOAT}:

  [DECLARATIONS]:

    [name_STRING]:name STRING:STRING(1..4}

    [pi]:pi:constant FLOAT := 3.141593

    name:=((("A", "B"),("C", "D")),(("E","F"),("G","H")))

    side2:=(-0.5,-0.5,-0.5, -0.5, -0.5, -0.5, -0.5, 0.5)

    Procedure section of original FORTRAN program.
```

Fig. 4 (b) The corresponding Ada Flexform after the FORTRAN had been reverse engineered and converted into Ada pseudocode.

```
PROCEDURE TEST   IS

done : BOOLEAN := TRUE;
name : array (1..2, 1..2, 1..2) of name_STRING;
side : array (1..5) of INTEGER;
side2 : array (1..8) of FLOAT;
name STRING:STRING(1..4);
pi:constant FLOAT := 3.141593;

BEGIN
   name:=((("A", "B"),("C", "D")),(("E","F"),("G","H")));
   side2:=(-0.5,-0.5,-0.5, -0.5, -0.5, -0.5, -0.5, 0.5);
   Procedure section of original FORTRAN program.;
END TEST;
```

Figure 4c: Ada source code generated from Ada Flexform.

In general, most FORTRAN-to-Ada complications reside in the data. These problems currently are usually handled during semantic postprocessing. The semantic postprocessor also deals with program- and system-level issues. For example, common variables in FORTRAN are global. In this case, the PRODOC FORTRAN-Ada Semantic Postprocessor puts these in an Ada Package Flexform, with appropriate references in modules using these resources. Similarly, FORTRAN Program Flexforms convert into top-level Ada Procedure Flexforms. The translated subroutines called therefrom are automatically inserted into Package Flexforms and corresponding Package Body Flexforms. Moreover, further reengineering can be accomplished automatically by building custom tools. Ada source code is generated automatically by simply selecting the proper option.

Producing a working Ada program, even one that is better than the original FORTRAN, still may not be enough. We might want to totally redesign certain parts of the new Ada program. The reengineering tool should support the creation of new designs by modeling intended functionality in the tool's high-level design language. Then a simulator can be used to debug the design. Successively moving between design and debugging ensures that you are always building on a solid foundation. Consequently, the need for source-code-level debugging is dramatically reduced, and the resulting systems are better designed. (This work has led naturally to a new cognitive [analysis, design and programming] paradigm, which generalizes on the object oriented paradigm. This work will be reported in a future article.)

Once a new design is acceptable, we convert it into Ada. Procedural code in high-level design Flexforms is automatically converted into Ada Flexforms. The final step involves linking the Ada Design Flexforms to reusable modules. The user must classify each identifier introduced in the original design (e.g., as either a variable or a function). In the case of variables, the user must also specify associated types in the reusable code.

The process is essentially the same with all legacy code. Code is reverse engineered, translated and semantically postprocessed. Optional redesign and/or customization comes next. Finally, the full Ada code is generated. There are, of course, special considerations. Unlike C, Ada does not support pointers to functions, so a custom strategy must be devised in C-Ada conversions. Similarly, structured COBOL typically requires explicit Exits (to avoid "fall-throughs"). These Exits are generally converted into Ada exceptions. Other languages require customization of both the syntax-oriented translator machinery and the semantic **post-processor.**

Case History: 600K LOC VAX Pascal to C++ or Ada

Problem: MacDonald, Dettwiler and Associates. Ltd. (MDA) is the world's leading supplier of commercial space remote sensing ground stations, installed in over 20 countries and capable of handling all major optical and radar imaging satellites. The company is also a major provider of advanced space-qualified software, air traffic control systems, defense electronics systems and network communications training and consulting.

MDA had developed a large satellite image processing system which was a core component to the Geo-Information Systems capabilities. The System had grown to over 1 million lines of highly optimized VAX Pascal code over its ten year development period. This system had become increasingly difficult to maintain under VMS and it was impossible to take advantage of the many new tools and libraries written in C/C++ and available under UNIX. Additional motivation came from a US military customer requiring delivery of their system on a UNIX workstation by a certain deadlline.

Options: Three options came to mind: 1) perform a source code-to-source code translation; 2) redo the entire system in the new language or 3) use a combination of re-engineering and translation.

As a target language, MDA had considered both Ada (because of its strong type checking) and C/C++. Toward this end, MDA might have enlisted the services of a company specializing in translations to move to the more modern language -- most likely to C or C++. Unfortunately, trying to convert complex, highly optimized code written in a very rich, highly nonstandardized language (VMS Pascal) on an increasingly obsolete operation system (VMS) into another language in another operating system is not a simple task. This problem was magnified because of low level operating system incompatibilities (e.g., differences in byte ordering). This first option presented the question: If the code is already hard to maintain on its intended platform, how easy will it be to maintain on a different platform in a different language with just a source to source translation?

The second option, to redo the software entirely in the new language cost more in time and money. MDA felt under tight time pressure to get this conversion completed.

The third option, the one MDA selected, was to perform a combination reengineering and translation. MDA wanted to modify the existing design, capture the reusable components of the existing systems and integrate everything onto the new platform in the new language. Part of the work requiring specific knowledge of the application was to be done "in-house." MDA needed help most in conversion of the actual code.

Evaluation: MDA requested 28 companies, both inside and outside North America, to quote on price and schedule to translate the image processing system. After initial evaluations, the three most promising company proposals were selected for intensive site visits. MDA sent a technical team to visit each of the three selected potential contractors, to meet their people and inspect the tools that had been proposed.

Selection: According to Lou C. Morena, Chief Engineer at MDA's National Remote Sensing Center Program, the contract was awarded for three main reasons:

1) The tool set was mature. Modification of the tool set to handle VAX Pascal was required, but the maturity and extensibility of the products to easily facilitate VAX Pascal conversion completely overshadowed the required customization effort.

2) The systematic approach proposed. including automation of the work, promised a chance of meeting MDA's very tight schedule while producing a predictable product.

3) Pricing was competitive with other proposals.

Process: MDA and contractor software engineers worked in parallel at their own sites. A new redesign effort was begun while the core legacy Pascal code was converted in C/C++. The basic idea was to identify and convert reusable portions of the VAX Pascal (over 600 K LOC. into C++ and to map the converted code into the new design.

Contractor personnel concentrated on the conversion effort. Unlike most conversion efforts -- that involve source to source translation -- the contractor's approach involved two major phases. During the first phase, all the VAX Pascal code was reverse engineered. In the second phase, the reverse engineered Pascal code was converted into C++.

The first phase involved reverse engineering the VAX Pascal code into a modular, object oriented Flexform repository where the underlying semantics were exposed. This step is extremely helpful in allowing the programmer both to obtain information about the system as well as the code and to automatically modify the system and the code.

During the second phase, the reverse engineered Pascal code was converted into C/C++. One month was scheduled to prepare detailed specifications for Pascal constructions, variant records, scoping rules and the like. Particulalry challenging were INHERITS, a wide variety of VMS attributes and VAX Pascal array and record extensions that required non-obvious C++ implementations. Once customized, the automated conversion took less than 6 hours -- over 100,000 lines of code per hour.

Working together, MDA and the contractor completed the entire conversion five weeks ahead of a highly aggressive schedule (five months). Automated conversion accuracy exceeded 99 percent -- without manual intervention. At a conversion rate of 100,000 lines of code per hour, it is hard to overestimate the significance of this technology to the Ada community.

Conclusions

The above methods and technologies clearly have direct application to perhaps the major dilemma facing DoD software managers as regards Ada. On the one hand, interoperability and maintainability strongly favor the use of Ada for mission-critical software. Budget constrains, on the other hand, make it impossible to redevelop in Ada even a modest fraction of the millions of lines of existing legacy code. The methods described herein demonstrate rather conclusively the dramatic cost reductions that would be possible if one were to systematically apply these methods. In this context, a first step might be to start with legacy FORTRAN code where technology is already in place. By employing the above methods, and fine tuning the automated conversion technology, this process could be expected to result in dramatic cost reductions. Having demonstrated such savings on a broad scale to critical legacy FORTRAN, this approach could then be applied with even greater confidence to other legacy code.

References

Boehm, B. W. A spiral model of software development enhancement. *IEEE Computer*, 1988, *21*, 61-72.

Martin, J. and McClure, C., *Action diagrams: clearly structured specification, programs and procedures.* Englewood Cliffs, NJ: Prentice Hall, 1988.

Miller, G. A. The magic number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 1956, *63*, 81-97.

Perry, W.E. Re-Engineering Splits Up Program and Its 'Owner'. *Data Processing*, 1989.

Scandura, J. M. A cognitive approach to software development: The PRODOC environment and associated methodology. *Journal of Pascal, Ada & Modula-2,* 1987 (Sept./Oct.), pp. 10-25. (Earlier Version in *Proceedings: First International Conference on Ada Programming Language Applications for the NASA Space Station.* University of Houston-Clear Lake & Johnson Space Center: June 2-5, 1986.)

Scandura, J. M. Cognitive approach to systems engineering and reengineering: integrating new designs with old systems. *Journal of Software Maintenance,* 1990, *2,* pp.145-156, *(Versions also in Journal of Structural Learning, CASEWorld, CASE Trends* and *Programmer's Update).*

Scandura, J. M. Cognitive technology and the PRODOC re/NuSys Workbench™: a technical overview. *Journal of Structural Learning and Intelligent Systems,* 1992, *11,* pp. 89-126.

Scandura, J. M. Converting legacy code into Ada: a cognitive approach. *IEEE,* 1994, pp. 55-61.

Scandura, J. M. Deterministic Theorizing in Structural Learning: three levels of empiricism. *Journal of Structural Learning*, 1971, 3, 21-53.

Scandura, J. M. *Structural Learning I: theory and research.* NY: Gordon & Breach Scientific Pub., 1973.

Scandura, J. M. *Problem Solving: a structural/process approach with instructional implications.* NY: Academic Press, 1977.

Scandura, J. M., Durnin, J. H. and Wulfeck, W. H. III Higher-order rule characterization of heuristics for compass and straight-edge constructions in geometry. *Artificial Intelligence*, 1974, *5*, 149-183.

Scandura, J. M. Automating renewal and conversion of legacy code. *Software Engineering Strategies.* NY: Auerbach Publications, 1994, March/April, pp. 31-43. (Reprinted in *Handbook of Applications Development*, Auerbach, 1995. Similar version in *Scuola estiva: Engineering of existing software.* Bari, Italy: Dipartimento di Informatica Universita degli Studi di Bari, 1994, pp. 179-192.)

Scandura, J. M. Structural analysis: part 2: toward precision, objectivity and systematization. *Journal of Structural Learning*, 1984, *8*, 1-28.

Scandura, J. M. A cognitive approach to software development: the PRODOC system and associated methodology. *Journal of Pascal, Ada and Modula-2*, 1987, *6 (Sept.-Oct.)*, 10-25.

Scandura, J. M. "Cognitive approach to systems engineering and re-engineering. integrating new designs with old systems," *Software maintenance: research and practice,* 1990, 145-156.

Voorhies D. J. & Scandura, J. M. Determination of memory load in information processing. In J. M. Scandura. *Problem solving: a structural/process approach*. NY: Academic Press, 1977. pp. 299-316.

Yeh, RT. An alternative paradigm for software evolution. In P.A. Ng & R.T. Yeh (Eds.) *Modern Software Engineering*. NY: Van Nostrand, 1990, pp. 7-22.

For More Information

SCANDURA-Flexsoft International, LLC
1249 Greentree
Narberth, PA 19072
(610) 664-1207 Voice
(610) 664-7276 FAX
jms@pobox.upenn.edu
http://www.scandura.com

PRODOC, FlexSys, FLOWform and Flexform are trademarks of Intelligent Micro Systems, Inc.