Automating Renewal and Conversion of Legacy Code

A Cognitive Approach

Joseph M. Scandura

Saddled with large quantities of obsolete but essential COBOL or FORTRAN code, large corporations and government agencies, including Department of Defense (DoD) contractors, face a series of unpalatable choices for dealing with this code. One option is simply to continue with the same old software and to patch it where possible to meet the most pressing needs. Properly implemented, either reengineering or redevelopment can add a degree of efficiency not previously attainable, and the recovery of costs from these efforts can often be accomplished within a reasonable period. The initial costs of such renewal efforts, on the other hand, are often prohibitive. In deciding what to do with obsolete code, DoD contractors must also deal with the complicated issue of coding renewed systems in Ada, which is required by the DoD.

This article proposes one solution for those who must resolve these issues. Current reengineering and conversion (e.g., language translation) processes are discussed. As part of this discussion, the role of the PRODOC reengineering systems, from Scandura Intelligent Systems, Inc., in these processes is examined. These systems are the basis for a new cognitive approach to systems engineering and reengineering, which can improve systems and minimize the costs of converting legacy code. This tool is used with the FLOWform representation format, also from Scandura Intelligent Systems, which allows systems engineers to better visualize

Joseph M. Scandura, PhD, is the director of the Structural Learning, Instructional Systems, and Computer-Based Instruction at the University of Pennsylvania, Philadelphia, and president of Scandura Intelligent Systems.
and therefore comprehend systems structure. FLOWforms can also be used to make systems modifications, which are also represented in an easily comprehended visual format. The use of PRODOC is illustrated in a case history and a description of continuing DoD-funded research on this cognitive approach.

A VIEW OF REENGINEERING TODAY

Several classes of reengineering tools have been developed in the past few years; these tools offer systems developers a multitude of choices. Reengineering is currently a hot area in software engineering. There is a great deal of confusion, however, about what reengineering involves and even more about available tools and their benefits. The following sections briefly review the strengths and limitations of tools for analysis, design recapture, and systems redesign.

Code Analysis

One class of tools performs code analysis. These tools provide complexity measures, calling hierarchies, cross-reference lists, and other information about the organization of code. Analysis tools help gather data on existing systems. This information is then referenced by programmers to help understand and modify code.

Clearly, code analysis yields positive benefits. Tools in this category, however, have a major limitation: they provide information about a system but cannot support the making of needed changes to the code. Thus, programmers can gain insight and information from automated code analyses, but they still have to find the source of the problems and ways to fix them. Other tools (e.g., editors) are needed for this purpose.

Restructuring Tools

Certain types of code modifications can be performed with automated tools. Commercially available tools exist, for example, to restructure FORTRAN and COBOL source code automatically. Various kinds of parsing and text string manipulations are generally used for this purpose.

Restructuring is almost always beneficial for FORTRAN programs. Eliminating GOTOs can significantly aid programmers in understanding code. Instead of having to scan and integrate scattered code fragments, programmers can easily see the relevant processes, because the restructuring tools have put them in one place.

The restructuring of COBOL is not done as easily, because GOTOs in COBOL cross module (i.e., paragraph) boundaries. The elimination of GOTOs and fall-throughs usually results in what are basically new COBOL programs. It has been observed that restructuring COBOL often results in divorcing the code from its owner. Nevertheless, if the goal is long-term maintainability or the conversion of Ada, COBOL restructuring is an essential first step.

Design Recapture

A second, somewhat newer class of tools is made for design recapture, which is the analysis of source code to determine and visually represent relationships between source code modules. Typically, the information obtained is represented in a structure chart or another type of module hierarchy diagram. The basic technology of design recapture usually involves simple parsing techniques, in which modules are identified and attendant relationships captured for making the visual representation. The process is not unlike extracting a table of contents from a book—that is, searching for headings and similar kinds of information and extracting that information from the body.

Extracting relationships within a system and representing them in a visual manner, which allows them to be more easily modified, is clearly worthwhile. Unfortunately, the structure of relationships cannot be relied on for making code changes, especially when the existing code is poorly designed.
Most reengineering tools provide limited access to module code. Some can also support editing tools that can modify such code. Despite these capabilities, these tools do not solve the biggest problem—understanding details of the code so that programmers can modify the code. The kinds of representations (e.g., bubble charts) used to represent high-level designs simply cannot be applied effectively 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, approximately 90%, obtaining an understanding of code, and the remaining 10% of the time making changes. Consequently, anything that facilitates such understanding can pay handsome dividends.

Pretty printing is one approach to facilitating this understanding. Simply printing code in a structured format often helps but does not solve the problem. Action diagrams help further by organizing code structure. They bracket code, thus making it easier to perceive groupings. The programmer, however, must distinguish different types of structures and separate relevant code from irrelevant detail. Action diagrams, for example, still contain extraneous syntax and bracketing information (e.g., ; ; BEGIN, END).

Studies have shown that irrelevant information, even when very familiar, increases cognitive strain. The greater the cognitive strain, the lower the capacity for productive thought. One attempt to reduce cognitive strain is FLOWforms, which visually distinguish different structures (e.g., sequences, selections, and loops) and display embedded structures hierarchically and in context.

The perceptual aspects of visual bracketing alone can improve comprehension. Exhibit 1b is more easily understood than Exhibit 1a because the surrounding lines are instantly perceived as distinct from the contents. The bracketing letters in Exhibit 1a are similar to the contents. Consequently, it takes explicit cognitive effort to distinguish the bracketing in Exhibit 1a from the essentials. If the diagrams in this exhibit were compressed vertically, the perceptual difference would have been even greater. The generalization to be drawn is that module visualization aids human comprehension both by representing structure visually and by eliminating irrelevant detail.

Although better visual representation helps, it is not sufficient. To be more effective, code analysis tools must automatically represent code visually and modify the code in the visual format. The PRODOC reengineering tool can automatically reverse engineer existing code into pseudocode FLOWforms. It currently does this for C, C++, Ada, Pascal,
FORTRAN, and COBOL code. Analyzing existing source code and representing it visually in this manner requires much more sophisticated parsing techniques than simply recapturing designs (i.e., relationships between modules).

The PRODOC designer provides a highly interactive environment where FLOWforms can easily be edited, documented, restructured, and customized to support multiple environments. The PRODOC pseudocode generator instantly regenerates full source as desired.

**Contextual Versus Separate Windowing**
Rectangles, circles, and triangles connected by lines (i.e., bubble charts) are most commonly used for representing structure charts and modules. They are, however, inadequate for many types of representation, especially of real-time systems, and they are hardly unique. For example, data flow diagrams, structure charts, control flow, entity relationship diagrams, and context diagrams can be represented in a variety of equivalent ways.

It is nonetheless important to distinguish various kinds of information to be represented. Each kind of information deals with a very different aspect of a system—module, module relationships, and file/unit relationships. Different types 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 can be said about modules or relationships between compilation units for files.

Separate windows are not desirable, however, for different levels of abstraction within the same type of representation. Abstract representation of data flow diagrams, for example, provides important contextual information that is useful in understanding the next level of detail. Displaying such information in different windows places severe restrictions on both the number of levels that can be displayed on a screen and the ease with which they can be understood. Expansion in different windows, for example, makes it difficult to remember which windows (i.e., expansions) go with which elements in other windows. Most experts agree that it is difficult to understand at one time more than three or four levels of a data flow diagram.

One solution to this problem is to use some form of contextual windowing. FLOWforms accomplish this by allowing explosion directly in context. Lower-level detail is automatically displayed within the element containing it. FLOWform rectangles can be expanded without affecting in any way the context above or below that in which they exist. This makes it possible to see more detail as well as the general picture. This is not possible by using such graphic elements as boxes or circles connected by lines. Attempting to open a visual element in this type of representation changes the overall scale. Consequently, the original context is quickly extended beyond the bounds of the monitor screen.

FLOWforms make it possible to view module relationships, unit relationships, and even program relationships at any desired level of abstraction. More detail can be viewed in one area without losing visual context.

Different kinds of representations (e.g., modules and call hierarchies) are represented in different windows. Although many tools allow switching between different views, FLOWforms differ because the same intuitive representation is used throughout. Unlike with other tools, the user never has to look at code, even at the very lowest levels of module detail. Using the PRODOC designer, all desired modifications are made directly in FLOWforms. There are no restrictions on this process. Anything that can be written in Ada, for example, can be written directly in Ada FLOWforms.
Conversion to New Environments

Hardware and operating systems are constantly being improved. Consequently, IS professionals are often faced with the task of converting old software for new environments. This conversion is complicated because of the need to maintain two or more versions of the old software simultaneously. Usually, separate teams of programmers maintain two or more sets of files. As changes are made or improvements are added, the files become increasingly different. An improvement would be to maintain multiple versions in one set of files.

This could be accomplished by using conditional compilation metacommands, which are supported by most compilers. The problem with this solution, however, is that it becomes increasingly difficult to read and understand program logic. Metacommands provide extra noise that, cognitive studies show, negatively affects human comprehension.

A better approach is to minimize the amount of irrelevant information to be processed. This is accomplished in FLOWforms by simply labeling structures that are unique to a given platform or operating system. These labels are referenced by PRODOC during code and report generation. This approach is currently supported for C, C++, Ada, Pascal, FORTRAN and COBOL, and makes it possible to support multiple versions or environments in one set of files.

Conversion Between Languages

In moving to a new operating environment (e.g., from MVS to UNIX), it is often desirable to convert from an existing programming language into a newer one, such as C++ or Ada. Some have argued that the only reasonable way to accomplish this is to rewrite the code. One obvious limitation of this approach is cost. A less obvious limitation perhaps is that the Ada code produced by the level of programmer likely to be assigned to the task might not be much better than the original.

A less costly approach involves the use of source-to-source translators. These tools convert source code from one language (e.g., FORTRAN or COBOL) directly into another (e.g., C, C++, or Ada). A common complaint about such tools is that they result in poor code (i.e., "AdaTRAN" or "AdaBOL").

A more effective approach is to convert the source language into an intermediate form that has semantic as well as syntactic characteristics. Code in the target language is generated from this intermediate language. Higher-quality code can often be achieved with this technique because the approach generally makes it easier to deal with semantic as well as syntactic issues. The overall process, however, is usually slower and more complex. High-level reengineering issues are also usually not solved.

A third approach, which is used by PRODOC, further refines the intermediate language approach by sharply distinguishing syntactic and semantic aspects of a translation. Thus, efficient parsing technologies can be used to complete syntactic aspects of the conversion rapidly; semantic issues are resolved by more powerful and typically slower semantic transformations. The result of such a conversion can approach and often exceed what one might reasonably expect from an expert human programmer.

There are, of course, practical limits to what any generic translation tool can do. Especially with older languages, language definitions and compilers as well as operating systems and libraries come in many variations. Consequently, a full solution to the conversion problem must allow for customization.

With PRODOC, automatic conversion is accomplished in two steps. The first is the reverse engineering of the source code (e.g., FORTRAN, COBOL, or C) into PRODOC's modular, object-oriented FLOWform repository. (This modularity makes the FLOWform repository suited for the client/server model.) Once reverse-engineered, the semantics and syntax of
the source code are directly accessible. The second step involves conversion. Visual FLOWforms containing pseudocode in one language are converted into pseudocode FLOWforms of another language. Parsing techniques perform the simpler syntactic conversion; a C or Ada semantic postprocessor takes the output of the PRODOC translator and turns it into good C or Ada.

Both the syntactic and semantic post-processing of a PRODOC conversion can be customized. The basic technology can easily be extended to such new languages as Jovial. Currently, C and Ada conversions from FORTRAN, C, Pascal, and COBOL are fully supported. Depending on the source language, 90% to 100% of the code is converted automatically, and higher-level designs are preserved in the process. Optional customization is available as desired.

The only significant constraint on semantic customization is the requirement to describe at a high level of abstraction exactly what is to be done. Implementation is straightforward once the requirements have been specified.

Customization is accomplished using PRODOC's semantic tool construction library and higher-order programming facilities. The library provides very high level building blocks suited for detecting patterns and manipulating system semantics. It is, in effect, a very high level object-oriented language that can semantically manipulate and construct FLOWforms. Any reengineering modification that can be described can be readily implemented using the library. In most cases, attempting the same kinds of manipulations through C or Ada programs operating directly on source code (e.g., ASCII files) would be quite impractical. Reverse-engineered and translated code is immediately available in the PRODOC designer's interactive, fully modifiable FLOWform environment.

Source-to-source and intermediate language translators are recommended if no further maintenance on the code is desired. This, however, is rare; if this were the case, why translate the code to begin with? The purpose of translating code is for better maintenance. It is widely recognized by the DoD, for example, that Ada programs are easier to maintain than programs written in FORTRAN, COBOL, C, or Jovial.

Systems Redesign

Although the previously discussed processes contribute to overall maintainability, the desire for continual enhancement implies the need for conversion capabilities that make explicit provisions for reengineering. Many situations call for creating an entirely new or renewed design. Rather than having to build an entirely new system, however, systems developers can, in most cases, salvage a considerable portion of the reverse-engineered or converted code. Reusable code may be either highly specific or relatively comprehensive. In most cases, however, it should be highly modular.

Systems Reuse. Perhaps the major advantage of reusing code to build a better system is that large, high-level modules can often be reused in implementing renewed designs. Experience suggests that a minimum of 50% of existing code can be reused for redesigned systems. The key to reusability is not only the quality of the code. Code reliability and stability over time are often more important. As long as the code works and is not likely to change, there is little need to tinker with it.

Most front-end CASE tools support the creation of new designs. Some also support simulating display and input screens, largely to ensure user satisfaction. Both of these capabilities (i.e., design and displaying user screens interactively) play an important role in systems design or redesign. Their roles are not, however, the only ones that matter. Confidence in a new, high-level design comes only from testing and debugging the underlying logic.
Unfortunately, testing is an expensive and time-consuming task, even with the assistance of automated testing tools. The standard approach is to test units as they are completed and then proceed to integration testing, which is a long, involved process. A fundamental, unsolvable problem with this approach is that it is impossible to test all paths, even equivalent classes of paths.

The number of tests required increases exponentially with complexity if all testing is done after implementation. Conversely, the number of tests required increases only additively if testing is done from the top down. For example, the number of empirical tests required for a rather simple system is on the order of $2^{100}$ when those tests are not conducted until after implementation is complete. On the other hand, 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. Unfortunately, executable specifications require learning an entirely new language, which can hinder overall benefits. Formal specification languages are required because commonly used design methodologies favor either data analysis (e.g., information engineering) or process analysis (e.g., structured analysis). Lacking a balanced approach to data and process, they do not lend themselves to debugging designs.

A cognitive approach to systems design requires that data and processes be considered in parallel. This, in turn, leads to specifications that may be both arbitrary and abstract 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 processes both be represented at the same level of abstraction. The PRODOC simulator deals explicitly with abstract specifications of this sort and consequently supports design debugging, which does not require learning a new language.

**Interfacing Renewed Designs**

Creating a high-level design and testing it, of course, is only one part of the problem. Another involves the interface between high-level designs and reverse-engineered and reusable code. One solution is to convert high-level designs to the target language and to create an interface between converted designs and the reusable code.

The PRODOC re/NuSys Workbench supports such a process. High-level designs are first translated automatically into Ada pseudocode FLOWforms using the PRODOC high-level-design-to-pseudocode converter. Checking processes built into the PRODOC designer provide an interactive, semiautomatic way to create links between converted designs and the data and processes referenced in those designs. Ada source code generated from such FLOWforms can be compiled and linked directly to the reusable code.

**Automating Systems Redesign**

The semantic tool construction library can be used to manipulate systems semantics in arbitrary ways under program control. For example, calling hierarchies usually include references to corresponding parameters. But what about global variables? Or routines exported from one file or compilation unit to another? To resolve these issues, a cost-effective way to create customized representations is needed. The semantic tool construction library provides a solution. Semantic tools based on the library can easily be modified or built to meet special requirements.

**A COGNITIVE APPROACH TO SYSTEMS RENEWAL**

Implicit in the previously discussed approaches to systems reengineering is an integrated cognitive approach to systems reengineering and conversion. This
Cognitive approach involves modeling and testing a system's structure and functions at a high level of abstraction as well as at lower levels until all available data and computational resources have been analyzed. The process is essentially the same whether structural analysis (i.e., the cognitive technology) is used to design and develop new systems or to reengineer old ones. For new systems development, the to-be-developed system exists only in the mind of the analyst, designer, and end user. For reengineering, this process begins with a fully functioning system. In both cases, heavy use is made of reusable routines and code is salvaged as a result of reengineering.

Conversion to Ada involves the physical representation of a system. The overall process is depicted in Exhibit 2.

As a result of ongoing maintenance, most software systems become spaghetti code (see Exhibit 2a). The individual modules of most systems can be reused, even though the whole system is rather disorganized. Most new design ideas also start out looking like spaghetti code.

As shown in Exhibit 2b, this code from the existing system is reverse-engineered into modular FLOWforms that make understanding module structure
easy and make needed repairs conveniently. This step, however, is unnecessary in developing new systems.

The desired system is redesigned at a high level, using PRODOC’s 4GL. The design is tested for logical errors, through simulation of process and data. At this point, a hierarchically structured system is created (see Exhibit 2c). This step is optional where the original structure is acceptable.

Reusable modules from the old system are then mapped into the new high-level system design. Finally, PRODOC is used to design and develop missing low-level routines (see Exhibit 2d). As much as 50% to 90% of existing modules are reusable.

Systems are automatically reengineered, multiple platforms are converted, and FLOWforms are converted to more modern languages through the use of custom reengineering tools and designer, translator, and semantic postprocessor options.

Introducing major enhancements did not seem possible.

Although 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 FLOWforms. (Currently, PRODOC requires half an hour to reverse-engineer 50,000 lines of code. The rudimentary version available in 1986 required about three weeks to do the same.)

The structure of the pseudocode still mirrored the original twisted design. Still, the visual representation of that design made the overall organization clearer, and the enhancements that had seemed impossible became at least seemingly possible.

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 system’s functions and the code organization, however, were largely out of sync. To clarify the system, FLOWforms modeled the system’s functions. The design was tested dynamically for verification purposes.

It was decided at this point that instead of salvaging the original code, a new design would be created and the lower-level routines were mapped to the design. Besides renaming variables, the entire mapping process required only two and one-half days of debugging. It is important to emphasize, however, that the new design had been carefully planned and tested, and the lower-level modules reused were fully understood.

A Short Case History
Motivated by the results of 20 years of basic research in cognitive and computer sciences, development of PRODOC was begun as a research project in 1983. By early 1984, PRODOC was increasingly used in its own development. Since 1985, all further development and enhancement has been done exclusively with PRODOC itself. By mid-1986, some bootstrap routines had been reverse-engineered into PRODOC.

The reverse-engineering of these bootstrap routines was the beginning of the cognitive approach to reengineering. There were almost 100,000 lines of bootstrap code; approximately half of them related to PRODOC’s dynamic simulation and prototyping facilities. This code was functioning correctly and it was doing essentially what was wanted. Like most production code, however, the code had gone through many revisions and parts had become very fragile. It was a chore to make even the most trivial changes for fear of introducing unanticipated interactions.

Application of the Methodology and Maturation of the Tool Set
After years of development, refinement, and beta testing, the PRODOC re/NuSys Workbench came of age in 1992. Since then, it has been used in a wide variety of situations, ranging from the simulation
any software engineer can readily understand them.

Further benefits derive from the ease with which structures can be collapsed and expanded in context. It is not necessary to consciously remember which expansion goes with which bubble in which higher-level bubble chart (i.e., separate windows). Consequently, there is no cognitive limit on the number of levels of refinement that can be understood. 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." Placing information in context effectively reduces the number of chunks to which a person must attend.

The results of passing the FORTRAN FLOWforms (see Exhibit 4) through the PRODOC translator are shown in Exhibit 5. In this example, the FORTRAN code is so simple that the translator can do all of the work directly. It can also handle more complex procedural constructions. For example, PRINT statements in FORTRAN are keyed to associated but possibly distant FORMAT statements. Both are needed to determine the corresponding Ada output statements.

In most cases, however, semantic postprocessing is a necessity. FORTRAN arrays, for example, can be initialized in DATA statements that are separate from variable and type declarations. These are defined in Ada as arrays but with the initialization in the procedural section of FLOWforms. Sample FORTRAN and the corresponding Ada FLOWform and source code are shown in Exhibit 6.

Most complications in translating from FORTRAN to Ada arise from the data. In PRODOC, these are handled almost exclusively during semantic postprocessing. The semantic postprocessor also deals with program- and system-level issues. For example, common variables in FORTRAN are global and may be used in any number of subroutines or functions. Consequently, the PRODOC FORTRAN-Ada semantic postprocessor places these in an
Ada package FLOWform, with appropriate references in modules using these resources. Similarly, FORTRAN program FLOWforms convert into top-level Ada procedure FLOWforms. The translated subroutines are automatically inserted into package FLOWforms with their respective bodies. Moreover, any further reengineering can be accomplished automatically by building custom tools with the semantic tool construction library.

The final step, of course, is the generation of the Ada code. This is accomplished automatically by simply selecting the proper option. Nonetheless, producing a working Ada program, albeit one that is better than the original FORTRAN, still may not be enough. Certain parts of the new Ada program may have to be totally redesigned. PRODOC also supports this process. The PRODOC Designer can be used to create a new design in PRODOC's high-level design language (i.e., Library FLOWforms), which models the intended function. Then, the PRODOC simulator is used to debug the design. Successively switching between design and debugging ensures a solid foundation. Consequently, the need for source code-level debugging is reduced, and the resulting systems are better designed.

Once a new design is acceptable, it is coded into Ada. The PRODOC translator greatly facilitates this task. Procedural code in Library FLOWforms is automatically converted into Ada pseudocode FLOWforms.

The final step involves linking the Ada design FLOWforms to reusable modules in the originally converted code. This is accomplished with PRODOC's checking process. Each identifier introduced in the original design is detected. The user must first classify it as either a variable or a function. In the case of variables, associated types must also be specified in the reusable code.

The process is essentially the same with all legacy code. The code is reverse-engineered, translated, and semantically postprocessed. Optional redesign or customization occurs next. Finally, the full Ada code is generated. There are, of course, special considerations. Ada simply does not support pointers to functions, for example, as does C. Hence, a custom strategy must be devised to handle this situation in C-to-Ada conversions. Structured COBOL typically requires explicit exits (to avoid fall-throughs). These exits are generally converted into Ada exceptions. Dealing with other languages, such as Jovial or CMS2, requires customization of both the syntax-oriented translator machinery and a customized semantic postprocessor.

CONCLUSION

This article illustrates how an existing system in FORTRAN, COBOL, or C can be restructured, reverse-engineered, and converted to Ada or C. How such a system could be reengineered is also examined. An existing or converted system might be redesigned in a neutral high-level design language, tested, and converted automatically to Ada, C, or any other language supported by PRODOC. Also, a new high-level Ada or C design can be linked to reusable Ada or C modules.
Ongoing Research

Conversion accuracy with PRODOC is currently approaching, and in some cases exceeding, 99%. At a conversion rate of 100,000 lines of code per day, the PRODOC tools used with the cognitive approach can benefit both the commercial and the Ada communities. The feasibility of the tools and approach has been demonstrated in Phase I research, and Intelligent Micro Systems, Inc., has been awarded a DoD contract to field-test and compare the methodology with current methods.

The principal investigator in this research is Alice B. Scandura. She is currently evaluating projects for their suitability. Participants in the research are trained in the cognitive approach and use of PRODOC and receive free consultation. Selection criteria are:

- A suitable reengineering or conversion project.
- A committed team.
Project visibility.

Anyone interested in participating can contact Alice Scandura at:

Scandura Intelligent Systems
822 Montgomery Ave., Suite 317
Narbeth PA 19072-1937.


Notes


