Converting Legacy Code into Ada: A Cognitive Approach

Joseph M. Scandura
University of Pennsylvania and Intelligent Micro Systems, Inc.

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. 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 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.

The visual bracketing that is provided for different structures in flowforms can improve comprehension. To see this, contrast Figures 1a and 1b. The surrounding lines 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, such as Prodocc, automatically reverse engineers existing code into pseudocode flowforms. The tool should provide an interactive environment where flowforms can be edited, documented, restructured, and customized to support multiple environments. In turn, the pseudocode generator regenerates full source code as desired. For example, Prodocc currently does this for C, C++, Ada, Pascal, Fortran, and Cobol.

**Contextual versus separate windowing.** Two kinds of representation are implicitly described above: representation of 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. Scandura, for example, shows how dataflow diagrams, structure charts, control-flow and entity-relationship diagrams, and context diagrams may be represented in flowforms.

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. Flowforms accomplish this by allowing "explosion" directly in context. Lower level detail is automatically displayed within the element that contains it. Thus, flowform rectangles can be expanded without affecting the context above or
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 flowform 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.

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."4,5)

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

Of course, different kinds of representations (for example, modules or call hierarchies) are represented in different windows. Flowforms 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 flowforms without restriction. Any program that can be written in Ada, for example, can be written directly in Ada flowforms.

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 flowforms 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 flowforms need to be preserved. This approach is currently supported for C, C++, Ada, Pascal, Fortran, and Cobol in the Prodoc environment.

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 "Adab," 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 Common 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 flowform repository. (This modularity makes the flowform repository ideally suited for the 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 preprocessing may be necessary with nonstandard code.)

The second step involves conversion. Visual flowforms containing pseudocode in one language are converted into flowforms containing pseudocode in another language. Parsing techniques perform the simpler syntactic conversions. For example, 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 result 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. Customization also is available. The only significant constraint is that you must be able to describe exactly what is to be done.

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.

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

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 highly 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.

As Scandura demonstrated, the number of tests required increases exponentially if all testing is done after implementation. Conversely, the number of tests only increases additively if testing is done from the top down. Approximately 100 empirical tests are required in the example cited when one waits until complete implementation before testing. Only about 500 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 pro-
cess both be represented at the same level of abstraction. The Prodoc Simulator deals explicitly with abstract specifications of this sort and does not require learning a new language.

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. One solution is to convert high-level designs to the target language and to create an interface between converted designs and the reusable code. Prodoc, for example, automatically converts high-level designs into Ada pseudocode flowforms. Checking processes in turn provide an interactive, semi-automatic way to create links between converted designs and the data/process resources referenced in those designs. Ada source code generated from such flowforms can be compiled and linked directly to the reusable code.

Automating system redesign. The above process provides strong support for renewing old systems. There are other possibilities as well. For example, the semantic tool construction facility within Prodoc can be used to manipulate system semantics in arbitrary ways under program control. Consider a simple example. Call hierarchies usually include references to corresponding parameters. But what about global variables or routines exported from one file or compilation unit to another? Semantic tools based on the STC Library can be easily modified (or built) to obtain such information.

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. 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. 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. Testing and diagnosis of individual knowledge at higher levels of abstraction is more efficient because fewer paths are involved.

Testing and debugging software designs makes sense only when data and process are represented at the same level of abstraction. “Structural analysis” with its emphasis on process is inadequate because it focuses attention on 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. As Scandura has shown, operations within objects are not the same as operations on objects. The former correspond to automatic human perception, the latter to conscious cognition. It will suffice here to call attention to the method of “structural analysis” (the “al” is intended). First used to construct representations of human knowledge, structural 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. The overall process and an early case history are described elsewhere.

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

Application of the methodology and maturation of the toolset. Reengineering tools have been used since the early 1980s in situations ranging from the simulation of new hardware designs (before implementation) to the conversion of legacy Fortran, C, and Cobol into C/C++ or Ada. Most applications, however, have involved reengineering and/or conversion.

The 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 flowform. 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 flowform structures directly in context.

An example of passing the Fortran flowform (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 Fortran statements. Both are needed to determine the corresponding Ada output statements.

In most cases, semantic postprocessing also is a necessity. Fortran arrays, for example, can be 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 and the corresponding Ada flowform and source code.

In general, most Fortran-to-Ada complications reside in the data. These problems are usually 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. In this case, the Prodoc Fortran-Ada Semantic Postprocessor puts these in an Ada Package Flowform, with appropriate references in modules using these resources. Similarly, Fortran Procedure Flowforms convert into top-level Ada Procedure Flowforms. The translated subroutines called therefrom are automatically inserted into Package Flowforms and corresponding Package Body Flowforms. Moreover, further reengineering can be accomplished automatically by building custom tools with the STC facilities. 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.

Once a new design is acceptable, we convert it into Ada. Procedural code in high-level design flowforms is automatically converted into Ada Flowforms. The final step involves linking the Ada Design Flowforms to reusable modules. The user must classify each identifier introduced in the original design (for example, choosing 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.

Conversion accuracy approaches and in some cases exceeds 99 percent. At a conversion rate of 100,000 lines of code per day, it is hard to overestimate the significance of this technology to the Ada community. We are now evaluating projects for further testing.
Figure 4. (a) Sample Fortran with a variety of data constructions; (b) Ada Flowform resulting from translation and semantic postprocessing; (c) Ada source code generated from Ada Flowform.

References


Joseph M. Scandura is Director of Instructional Systems at the University of Pennsylvania. His research interests include structural learning and computer-aided systems engineering. He founded Intelligent Micro Systems, Inc., a firm specializing in software engineering research and development and services. Scandura has a PhD in mathematics and education from Syracuse University, and MA and BA degrees from the University of Michigan in educational psychology and mathematics, respectively. He is editor-in-chief of the Journal of Structural Learning and a fellow of the American Psychological Association.

Readers can contact the author at Intelligent Micro Systems Inc., Suite 317, 822 Montgomery Ave., Narberth PA, 19072-1937; e-mail jms@pobox.upenn.edu.

April 1994

61