AUTOMATING RENEWAL AND CONVERSION OF LEGACY CODE INTO ADA: A Cognitive Approach Using PRODOC

Joseph M. Scandura, Ph.D.
University of Pennsylvania and
Scandura Intelligent Systems
822 Montgomery Ave., Narberth PA, 19072-1937

Abstract -- This paper reviews the current status of the re-engineering industry and proposes a cognitive approach to system maintenance which can dramatically reduce Ada renewal costs. This cognitive methodology 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 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 code salvaged as a result of re-engineering. Conversion to Ada pertains largely to how a system is physically represented.

This methodology is supported by the PRODOC design, development and maintenance system. Results to date indicate that legacy code can be re-engineered and/or converted into Ada (or C/C++) in large part automatically, and at greatly reduced cost. Finally, a call is issued for participants in Phase II SBIR research.

Index Terms -- Design, Re-engineering, Conversion, Translation, COBOL-Ada, FORTRAN-Ada, C-Ada, Reuse.

Saddled with large quantities of obsolete but essential COBOL or Fortran, DoD contractors and DoD agencies are faced with a series of unpalatable choices. One option is to simply continue with the same old software, patching it where possible to meet the most pressing needs. Properly done and implemented, either re-engineering or redevelopment can add a measure of efficiency previously impossible -- and cost recovery can often be accomplished over a reasonable period of time. The initial costs involved in such renewal, however, are often prohibitive. Decision making is further complicated by DoD directives to convert systems requiring significant change into Ada.

Faced with this dilemma, what are decision makers to do? In this paper, I first review the current status of the re-engineering and conversion (e.g., language translation) industry. Then, I describe the genesis of a new cognitive approach to system engineering and re-engineering (Cognitive Approach) which offers a solution to this dilemma. Next, I show how the Cognitive Approach can dramatically improve systems and minimize the costs of converting legacy code into Ada. Finally, I present an example and describe ongoing Phase II SBIR research.

1. RE-ENGINEERING: CURRENT STATUS

Several classes of re-engineering tools have evolved over the past few years which appear to offer an array of choices. Indeed, re-engineering (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 re-engineering involves, and even more so as to benefits of the kinds of re-engineering tools that are currently available. In this section, I briefly review analysis, design recapture and system redesign tools, along with major strengths and limitations of each type.

Code Analysis -- One class of tools involves the analysis of code. Analysis tools are used to provide complexity measures, calling hierarchies, cross reference lists and other information concerning the organization of code (or the lack thereof). Analysis tools help gather information about existing systems. This information is then referenced by programmers to help understand and/or modify code.

Clearly, code analysis yields positive benefits. Tools in this category, however, 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 and/or information 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 largely be automated. Commercially available tools exist, for example, to automatically restructure Fortran and COBOL source code. Restructuring is almost always desirable in the case of Fortran. Eliminating GOTO's significantly facilitates code understanding from a cognitive point of view. Instead of having to scan and integrate scattered code fragments, immediately relevant processes are all available in one context. COBOL restructuring poses a unique set of problems, however, because GOTO's cross module (paragraph) boundaries. Eliminating GOTO's 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 -- 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 other module hierarchy.

The basic technology generally involves simple parsing techniques in which modules are identified and attendant relationships 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 re-engineering tools provide limited access to module code. Some can also 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 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 (estimated in the area of 90%) understanding code and only 10% making changes. Consequently, anything which can facilitate such understanding will pay handsome dividends.

"Pretty Printing" is a step in this direction but is not sufficient. Action diagrams (e.g., Martin 1988; Scandura, 1990) help further by organizing code structure. Action diagrams bracket code, making it easier to perceive structure groupings. It is still left to the human, 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 (e.g., ], }, ; BEGIN, END, etc.). 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. FLOWforms (Scandura 1987, 1990) provide significant help by visually distinguishing different structures (e.g., sequences, selections, loops), and displaying embedded structures hierarchically and in context. The perceptual aspects of visual bracketing alone can dramatically improve comprehension. To see this, contrast Figs 1A and 1B.

![Fig. 1A. FLOWform with indistinguishable borders](image1.png)

![Fig. 1B. FLOWform with perceptually clear borders](image2.png)

The reason for this improvement is that the surrounding lines in Fig. 1B are directly and automatically perceived as distinct from the contents. The bracketing letters in Fig. 1A are of the same genre. Consequently, it takes explicit cognitive effort to distinguish the bracketing from the essentials. If the above diagrams had been 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 improve on code analysis tools it must be possible: (a) to automatically construct such visualizations from code and (b) to modify the code directly in the visual environment. In this context, the PRODOC re-engineer automatically reverse engineers existing code into pseudocode FLOWforms. It currently does this for C, C++, Ada, Pascal, Fortran and COBOL. The PRODOC Designer provides an interactive environment where FLOWforms can be
edited, documented, restructured and customized to support multiple environments. In turn, the PRODOC Pseudocode Generator regenerates full source code as desired.

**Contextual vs. 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 (1992), for example, shows how data flow diagrams, structure charts, control flow, 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: module, module relationships, file/unit relationships. 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 relationships between compilation units or files.

Separate windows are not desirable, however, when one is talking about different levels of abstraction *within* the same type of representation. Displaying such information in different windows places severe restrictions on both the number of levels which 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 (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 data flow 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 which contains it. Thus, FLOWform rectangles may be expanded without affecting the context above or below. This makes it possible to see more detail without losing the general picture. Graphic elements (e.g., 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.

FLOWforms make it possible to view module relationships, unit relationships, and even program relationships at any desired level of abstraction.

![Diagram](image)

**Fig. 2. This diagram is reproduced from U.S. 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 (e.g., a dotted frame). "Fanning in" (as well as "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 (e.g., modules, call hierarchies) are represented in different windows. FLOWforms differ from more eclectic views in that the same intuitive representation is used throughout. Unlike tools which use other representations, one never has to look at code per se, even at the very lowest levels of module detail. Using the PRODOC Designer, all desired modifications are made directly in FLOWforms with no restrictions. 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 meta commands supported by most compilers. This approach, however, clutters the code and makes it increasingly difficult to read and understand program logic. Meta commands provide extra "noise" which, as cognitive studies show, negatively impact human comprehension.
The amount of irrelevant information is minimized in FLOWforms by labeling structures which are unique to a given platform or operating system. These labels are referenced by PRODOC during code and/or report generation to automatically produce various versions on demand. Consequently, only the FLOWforms need to be preserved. 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 environment (e.g., from MVS to PCDOS 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 obvious limitation of 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. Another approach involves use of source to source translators. These tools 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 re-engineering issues also tend to go untouched.

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 (e.g., Fortran “format” and “print”) statements into the target language. Semantic transformations become increasingly necessary as the mappings address deeper (or more abstract) semantic differences (e.g., Fortran COMMON versus Ada packages).

What most differentiates this fourth approach is that the conversion works both syntactically from 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 which goes into making high level design decisions. The results of such conversions can approach or even exceed those of 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.

In PRODOC, automatic conversion is accomplished in two steps. The first involves reverse engineering the source code (e.g., Fortran, COBOL or C) into a modular, object-oriented FLOWworm repository. (This modularity makes the FLOWworm repository ideally suited for the client-server model.) Once reverse engineered, the semantics as well as the 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 containing another language. Parsing techniques perform the simpler syntactic conversions. Ada Semantic Postprocessors take the output of the PRODOC Translator and turn it into good Ada. “Good” in this context means that use is made of Ada constructs (such as Packages) which have no direct counterpart in the source language. It does not necessary imply that the result would be indistinguishable from that produced by an Ada expert.

Both the syntactic and semantic postprocessing aspects are customizable. The basic machinery also is extensible to new languages. Currently, C/C++ and Ada conversion, from Fortran, C, Pascal and COBOL are supported. From 90 to 99% of the code is converted automatically with higher level designs preserved in the process. Customization also is available. The only significant constraint is that one must be able to describe exactly what is to be done. Implementation using PRODOC’s Semantic Tool Construction Library (STCL) is straightforward. The STCL provides very high level building blocks which make it easy to detect patterns and manipulate system semantics. It is a very high level object oriented language designed for the semantic manipulation and construction of FLOWforms. Using the STCL, one can easily construct semantic tools to perform deep semantic manipulations otherwise possible only by

* Reverse engineering normally is fully automatic, although minor preprocessing may be necessary with non-standard code.
human intervention. In many cases, manipulating source code directly via C or Ada programs would be impractical.

Automation aside, some people want to do things their own way. To support such preferences, reverse engineered and/or translated code can be modified in PRODOC's interactive FLOWform environment.

To summarize, source to source and intermediate language translators represent a reasonable approach if no further maintenance on the code is desired. If so, however, why would one want to translate the code to begin with? The purpose of translation usually is 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.-- While the above capabilities contribute to overall maintainability, the desire for continuing enhancement implies the need for conversion capabilities which make explicit provision for re-engineering. 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 of the reverse engineered/converted code. Reusable code may 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, at a minimum, that 50 to 60% of existing code, and usually much more -- to over 99%, is reusable in redesigned systems. The key to reusability is not simply quality of code. Code stability over time can be even more important. As long as the black box 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 insure user satisfaction. Both of these factors (i.e., design and displaying user screens interactively) play an important role in system design or redesign. They are not sufficient, however. 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 painful process. A fundamental, unsolvable problem with this approach is that it is impossible to test all paths, even equivalence classes of paths (cf. Scandura, 1971, 1973, 1977). As Scandura (1990) demonstrated, the number of tests required goes up exponentially with complexity if all testing is done after implementation. Conversely, the number of tests only goes up additively if testing is done from the top down. The number of empirical tests required in the example cited, is on the order of $2^{100}$ 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. Executable specifications, unfortunately, often require learning an entirely new language -- a hindrance which can greatly reduce overall benefits. Formal specification languages are required largely because the 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 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. The PRODOC Simulator deals explicitly with abstract specifications of this sort, and does not require learning a new language.

Interfacing Renewed Designs.-- Creating a high level design and testing it 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 supports this process. High level designs are first converted automatically into Ada pseudocode FLOWforms. In turn, checking processes 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. The Semantic Tool Construction Library (STCL) 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 STCL can easily be modified (or built) to obtain such information. More generally, true semantic manipulation makes it possible to contemplate what has heretofore been unthinkable. For example, what if we had a tool that could take a poorly designed system -- one which, say, had dysfunctional packaging -- and automatically turn it into a well designed Ada program. Far out as this may seem, such tools are well within sight.

2. COGNITIVE APPROACH TO SYSTEM RENEWAL

Implicit in the above is an integrated cognitive approach to system development, re-engineering and conversion. This 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. An essential characteristic is that both data and process must be represented at the same level of abstraction. This is quite analogous to the representation of human knowledge (e.g., Scandura, 1971, 1979, 1974, 1977). The more knowledgeable the human population being modeled, the more abstractly can one usefully represent the knowledge in question. Put differently, the larger the “chunks” (e.g., Miller, 1956) or atomic rules (e.g., Scandura, 1971, 1979), the more easily any system (or body of knowledge) can be modeled. Testing and diagnosis of individual knowledge is more efficient due to the smaller number of paths involved (e.g., Scandura, 1971, 1979, 1977).

Testing and debugging software designs only makes sense when data and process are represented at the same level of abstraction. “Structured Analysis” with its emphasis on process is inadequate because it defers attention to 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.

It will suffice here to call attention to the method of “Structural Analysis” (the "all" is intended: e.g., see Scandura, 1974, 1982, 1984). 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 (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.

System renewal involves modeling 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 (e.g., 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 (Scandura, 1990, 1992, 1994).

Application of the Methodology and Maturation of the Tool Set -- Since its origins in 1983, PRODOC has been used 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 re-engineering and/or conversion.

![Fig. 4A. Sample Fortran source code.](image)
![4B. Fortran source code from A after restructuring and re-engineering into a FLOWform.](image)

![Fig. 5. Ada FLOWform resulting from Fortran-Ada translation.](image)
The Ada conversion process is shown in Fig. 4. Figure 4B 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. Further benefits in actual use derive from the ease with which one can collapse and expand FLOWform structures directly in context. In contrast to “bubble charts”, it is not necessary to remember which expansion goes with which bubble in which higher level window. 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” (e.g., Miller, 1956; Voorhies & Scandura, 1977). Putting information in context effectively reduces the number of “chunks” to which one must attend.

The results of passing the Fortran FLOWform (in Fig. 4B) through the PRODOC Translator is shown in Fig. 5. 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. PRINT statements in Fortran, for example, are keyed to associated but possibly distant FORMAT 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. Sample Fortran and the corresponding Ada FLOWform and source code are shown in Figs. 6a, b and c.

Fig. 6B. Ada FLOWform resulting from translation and semantic postprocessing.

In general, most Fortran to Ada complications reside in 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. Consequently, the PRODOC Fortran-Ada Semantic Postprocessor puts 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 called therefrom are automatically inserted into package FLOWforms with their respective bodies. Moreover, further re-engineering can be accomplished automatically by building custom tools with the STCL. Similarly, generating Ada source code is accomplished automatically by simply selecting the proper option.

Producing a working Ada program, albeit one which is better than the original Fortran, still may not be enough. We might want to totally redesign certain parts of the new Ada program. PRODOC also supports this process. The Designer is used to create a new design, which models the intended functionality, in PRODOC’s high level design, or LIBRARY, language. Then, the PRODOC Simulator is used to debug the design. Successively moving between design and debugging ensures that one is 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 the new design into Ada. The PRODOC Translator facilitates this task. Procedural code in LIBRARY FLOWforms is automatically converted into

Fig. 6A. Sample Fortran with a variety of data constructions.

6C. Ada source generated from Ada FLOWform.
Ada FLOWforms. The final step involves linking the Ada design FLOWforms to reusable modules via PRODOC's checking process. The user must first classify each identifier introduced in the original design (e.g., make a choice between, say a variable or a function). In the case of variables, one 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 come next. Finally, the full Ada code is generated. There are, of course, special considerations. Ada does not support pointers to functions, for example, as does C. A custom strategy must be devised to handle this situation in C-Ada conversions. 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 postprocessor.

To summarize, existing systems in Fortran, COBOL or C can be restructured, reverse engineered and converted to Ada (or C/C++). We have also described how such systems might be re-engineered. Then, we showed how converted systems might be redesigned in a neutral high level design language, tested and converted automatically to Ada. Finally, we showed how high level Ada designs can be linked to reusable Ada modules.

Ongoing Research -- Conversion accuracy approaches, even exceeds 99% in some cases. At a conversion rate of 100,000 LOC per day, it is hard to overestimate the significance of this technology to the Ada community. Having demonstrated feasibility in our Phase I research, Intelligent Micro Systems, Inc. has been awarded an SBIR Phase II contract to field test and compare the methodology with current methods. The principal investigator, Dr. Alice B. Scandura, is currently evaluating projects for their suitability. Participants will get training in the cognitive approach, use of PRODOC, and free consultation. Selection criteria are: (a) a suitable re-engineering or conversion project, (b) a committed team and (c) project visibility.

References:


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.


* At the current time, PRODOC does not include a LIBRARY-Ada Semantic Postprocessor. Hence, minor modifications might be needed -- for example, in the case of parallel processing.

** Interested parties should contact Dr. Scandura for details. She can be reached at (215) 684-1207.