A Cognitive Approach to Software Development: The PRODOC Environment and Associated Methodology

Joseph M. Scandura

University of Pennsylvania, 3700 Walnut St., Philadelphia, PA 19104

Most software development methodologies and tools fail to adequately bridge the gap between user defined needs (requirements analysis) and detailed program specifications (system design, e.g., [1], Ch. 5). Equally important is the need for integrated environments which support the software development process in a unified way. This report describes a cognitive systems (not necessarily AI) approach to software development. More specifically, it focuses on Intelligent Micro Systems, Inc.'s (IMS) PRODOC software development system, its underlying rational, and its associated methodology. PRODOC development was based on the assumption that application domain experts are best able to define tasks to be solved and how best to solve them, and that program designs should closely reflect expert-generated requirements. Traditionally, high maintenance costs are believed to result not just from poor design per se but from the failure of software designs to accurately reflect the "real world." Symptoms include poor communication between domain experts and software engineers, such as frequently large differences in perceived difficulty in making software modifications. PRODOC has been highly responsive, interactive MS-DOS PC-AT (640K RAM) environment. It provides a comprehensive but simple and uniform means of graphically representing data and process during all phases of the software development process: requirements definition, system design, prototyping, code generation, and maintenance. PRODOC is based on a cognitive (rule) construct, which employs a combined top-down, bottom-up approach in representing both objects (data) and procedures. Terminology used in creating PRODOC programs may be customized to match the way human application experts naturally describe the relevant data and operations. PRODOC supports programming in any language. In the case of Pascal, C, and Ada, PRODOC also provides full pseudo code support, including syntax and consistency checking, and automatic declarations and code generation. In this article, the cognitive construct (rule) on which PRODOC is based is precisely defined and represented in terms of visual Scandura FLOWforms. Then, PRODOC itself is described followed by an overview of the IMS System Development Methodology using PRODOC.

INTRODUCTION TO THE PROBLEM

All managers must exercise controls if they are to assure quality on time and within budget. In
PRODOC Environment and Methodology

addition to strictly managerial issues, however, software development poses many unique problems. The continuing need and expense of software maintenance, for example, makes quality control an important issue, not only with regard to initial products but over the long term as well. To make matters worse, failures to meet budget constraints and deadlines in software development have become almost endemic.

The software development process is normally broken into a number of distinct phases: planning, requirements analysis, design (including data and program structure, modules, etc.), prototype development, code generation, testing, and maintenance.

Software planning involves determining the scope of the software, needed hardware and human resources, time and cost estimations, and scheduling. Requirements analysis involves description of the information (data) and its structure, what is done to the information and what one gets as a result. These descriptions normally take a variety of verbal and/or graphic forms.

System design is undoubtedly the most important step in software implementation. Design involves translating and refining software requirements into progressively more detail. Mistakes at this level are invariably difficult to find and fix unless caught before actual coding begins. At this point, prototypes are sometimes developed but, more often, they are not. Programmers more typically start with design specifications and convert them to code in some standard programming language. Once coding has been completed, the program is tested and debugged.

The first two phases of software development have been the province of software managers, although those with computer science backgrounds often become involved with software design as well. Software coding, testing, debugging, and maintenance are typically handled by programmers.

Oddly enough, over 60% of the software development cycle takes place after the above steps have been performed (e.g., [2], p.322). Once “completed,” software is invariably changed in one way or another: to perfect its operation, to correct problems, to adapt the program to meet new needs, etc. Major changes in software are most expensive during this phase.

Available Methodologies and Tools

To improve productivity and minimize the resources needed for program maintenance, a wide variety of software development methodologies and diagramming techniques have been developed to support the process. These include DeMarco structured analysis, object-oriented design, structure diagrams, decomposition diagrams, Warnier–Orr diagrams, structure charts, and flow charts [1–3]. These methods and diagramming techniques provide the user with various perspectives on the to-be-developed software.

More importantly, given the complexity of large software systems, a wide variety of tools (or environments) have become available to assist with development.

(a) Various project management software packages are available to help track, control, and integrate progress.

(b) A number of tools also are available to assist managers with requirements analysis. These tools make it possible for users to maintain basic information about the software and how it is to be used in a central database, and to represent that information graphically, to present it in reports for documentation purposes, etc. Some even allow the development of sample screens for prototyping purposes.

(c) To make life bearable, a wide variety of tools have been developed to assist programmers in their work: languages which allow them to communicate in human decipherable form, editors to enter code, compilers and assemblers to convert code into machine readable form, etc.

A major limitation of current software development methodologies and tools is their failure to address the single most important issue in software development: how to bridge the gap between user-defined needs (which the software manager addresses in planning and requirements analysis) and how those requirements are implemented by the programmer (e.g., [1], Ch. 5). Undoubtedly, the single most important gap involves software design.

However, there is a serious lack of tools which adequately support the process of software design. Even more serious is the need for integrated environments, which support the entire software development process. The desirability of such environments is increasingly recognized but, unfortunately, no such environment currently exists.

A1-Based Research Tools

In addition to commercially available and/or widely used products, intensive research is underway at a number of institutions, based on expert systems technologies combined with natural language front ends. Among the prototypic software sys-
tems developed for this purpose are Gist from the University of Southern California’s Information Sciences Institute, Phi and Chi from the Kestrel Institute, and Programmer’s Apprentice from MIT. The original view in most of this research was that one could develop intelligent work stations that would do the whole job, from accepting free form English to the generation of verified code. Today, this seems unattainable using existing natural language and expert systems technologies. As in traditional approaches, the problem of software design appears most intractable. More immediately attainable, perhaps, is the development of knowledge-based inference tools (expert systems) which deal with various management and/or programming related tasks (e.g., Refine from Reasoning Corporation, the Crystal project at the Carnegie Group).

In an attempt to solve this program, considerable resources are being devoted to the further development of AI technologies. Still, the current consensus is that AI-based programming tools have a long way to go before they become practical. They require substantial computing resources and they are not particularly easy to use. As stated by Hindin [20] in a recent review, “It will be a long time before the (AI) tools will be as ubiquitous as the DEC VAX and Unix software development environments. The tools’ real value is that they are models for future work.”

Not everyone, of course, believes that natural language and/or expert systems techniques are the wave of the future. It’s now recognized, for example, that machine conversion of natural language into formal language is extremely difficult—and, in any case, may not be the best approach to take. Moreover, expert systems technology goes back to the 1960’s when a major AI technique involved the logical analysis of large state spaces. The resulting “combinatorial explosion” problem resulted in a shift in AI research in the early 1970’s toward the use of heuristics and clever programming techniques. Even here, many [4,5] have been concerned that the research has often been ideosyncratic, noncumulative, and more reflective of the programming language used (Lisp, Prolog) than the cognitive processes being modeled. There is a growing realization that, if it is to achieve its full potential, AI research must increasingly build on what has and is being learned about human cognition.

A COGNITIVE SYSTEMS APPROACH TO SOFTWARE DEVELOPMENT

This report describes a cognitive systems (not necessarily AI) approach to the software develop-

Structural Analysis

Over the past 20–25 years, a body of relevant knowledge has developed in structural learning, the science of cognitive, instructional, and intelligent systems engineering [5]. Most directly relevant is the research on structural (cognitive task) analysis, which is concerned with the systematic analysis of tasks/problems to be solved [6–10]. The results of structural analysis are precise representations of cognitive constructs, called “rules” (not to be confused with production rules; see later). Rules represent the knowledge needed for solving tasks in given problem domains.

Structural analysis combines top-down, bottom-up, and inductive approaches to knowledge representation (see esp. [9]). Initially partial and ultimately complete rule representations are induced from sample solutions (generated by domain experts) at each stage of top-down analysis. Analysis continues until each component corresponds to an elementary or atomic cognitive unit. Each major aspect of structural analysis has a direct parallel in software development: top-down analysis, bottom-up synthesis of existing library routines, and programming by example. Serious research has begun in a number
of these areas; [11,12]; however, much more needs to be done.

A long-range goal of IMS's research is full implementation of structural analysis. When complete, the resulting system will make it possible for domain experts to create computer programs by solving sample tasks interactively. The guiding philosophy in this research is to let the computer do what it might reasonably do, and require users to input only what they must.

PRODOC Overview

PRODOC constitutes a major step in automating structural analysis. It has been fully implemented, tested, and used extensively in a highly responsive, interactive MS-DOS PC-AT (640K RAM) environment (Unix conversion is currently under way).

PRODOC supports the entire systems software development process, from requirements definition and system design to prototyping, code generation, and maintenance. It employs a combined top-down and bottom-up approach to the representation of both objects (data) and procedures which operate on the objects. In addition, the terminology used in creating PRODOC programs may be customized so as to match the way human experts in any given application area naturally describe the relevant data and operations. This customized terminology is all based on a uniform, very simple syntax that can easily be learned (in at most a few minutes). The approach taken with PRODOC is general, as well as efficient and easy to use.

In other ways, the current version of PRODOC is not quite so ambitious. Rather than inputting sample solutions, software development involves users interacting with an unique, easy-to-use, and uniform visual interface. Thus, instead of "hiding" program structure, PRODOC represents such structure graphically using visual programming techniques [20,26].

In the next section, we define more precisely what we mean by a rule and show how rules can be represented as Scandura FLOWforms. Next, we describe the PRODOC system itself. Finally, we provide an overview of the IMS System Development Methodology using PRODOC.

THE RULE CONSTRUCT AND SCANDURA FLOWforms

Rules have three major components: a domain or set of data structures on which the rule operates, a range or set of structures which the rule purports to generate, and a procedure [13]. Rules have been shown to provide a convenient way to represent a wide variety of human cognitive processes, as well as arbitrary computer systems [7,14–18].

The term "rule" corresponds directly to the concept of a program. The "procedure" component of a rule (i.e., step-by-step prescriptions for carrying out the rule) corresponds directly to the procedural portion of a program. "Domain" and "Range" components of rules define problem schemas (i.e., classes of problems) and refer to input and output structures. Collectively, they correspond to the data structures on which programs operate. These correspondences are summarized below:

```
Program / \ / \ Data Structures Procedure

Rule / \ / \ Domain/Range (input/output) Structures Procedure
```

In general, the execution of rule procedures involves both testing conditions and carrying out operations. Where the internal structure of a rule procedure is unimportant, the rule is "atomic" or elementary—i.e., is viewed as indivisible. Those familiar with production rules will note that PRODOC rules are more general. The procedures of production rules consist solely of operations and, consequently, correspond to "atomic" rules.

In programming parlance, atomic rules correspond to program "subroutines," or in the case of PRODOC to "library rules." (As described below, PRODOC makes it possible to create libraries of rules which make it easy for nonprogrammers, as well as programmers, to construct programs.)

Rules as well as programs may be represented visually in terms of Scandura FLOWforms. FLOWforms appear similar to Nassi–Shneiderman flow charts but they make better use of the rectangular screen, are even easier to read, and allow simulta-
neous representation of as many (or as few) levels of refinement as desired. In addition, they can be used equally well to represent both arbitrary data and arbitrary procedures. The two types are distinguished by the term “data FLOWform” and “procedure FLOWform.”

Roughly speaking, a procedure or algorithm is a recipe, process, technique, or systematic method for doing something. More precisely, according to Knuth [21], a procedure or algorithm must:

1. always terminate after a finite number of steps,
2. include only definite steps that are precisely defined, with actions that can be carried out rigorously and unambiguously,
3. have an associated (possibly empty) class of inputs, or domain,
4. generate at least one output, and
5. be effective in the sense that all of the operations to be performed must be sufficiently basic that, in principle, they can be done exactly and in finite time by a person using pencil and paper.

Not all procedures are structured, however. Structured procedures are composed of substructures (components) or elements which have unique points of entry and exit. In order to ensure this property, each step in a structured procedure must be decomposable into one of three basic types of components:

(a) a sequence of steps or operations,
(b) a branch or selection from two or more conditional steps, or
(c) an iteration or loop

These types are illustrated below both in terms of traditional flowcharts and Scandura FLOWforms. In the former case, (a) the rectangles represent arbitrary operations (e.g., add a and b) and the diamonds represent (b) arbitrary selection or “if” conditions (e.g., IF the building is over 20' tall, then . . . ) and (c) arbitrary looping (“while”) conditions (e.g., While there is still further to go . . . ).

In Scandura FLOWforms, these three types of components are represented as shown following:

- Sequence
- Selection (IF . . . THEN . . . ELSE)
- Iteration (WHILE . . . DO)

These three basic types of decomposition are universally applicable and independent of any particular programming language (or any natural language for that matter). Moreover, used in combination via successive refinement, they have been proven adequate for any system design or programming task. Hence, there is no loss of generality in requiring that a procedure be structured.
PRODOC Environment and Methodology

Nonetheless, it is often convenient to allow certain variations on the above. Some common variations on selections and iterations are shown following:

\[
\begin{array}{c}
\text{CASE OF} \\
1 \\
2 \\
n \\
\end{array}
\]

Selection (CASE)

\[
\begin{array}{c}
\text{REPEAT} \\
\text{UNTIL} \\
\end{array}
\]

Iteration (REPEAT...UNTIL)

\[
\begin{array}{c}
\text{FOR} \\
\text{DO} \\
\end{array}
\]

Iteration (FOR...DO)

Other kinds of structures, of course, are sometimes used in particular programming languages. Pascal, for example, also supports a WITH (Record . . . Do) structure. Such structures usually can be represented suggestively in terms of the basic forms. The WITH structure, for example, may be represented as:

\[
\begin{array}{c}
\text{WITH record} \\
\text{DO} \\
\end{array}
\]

field variables

with (Pascal only)

In Scandura FLOWforms, sequence structures are often displayed using PRODOC with indentation to show level of refinement. This makes it easier to move about and otherwise manipulate FLOWforms on the screen. A sample FLOWform showing such indentation along with a variety of structure (decomposition) types follows:
Procedure FLOWforms may be recursive as long as the language in question supports recursion. This is certainly the case, for example, with Pascal, C, Ada, and Lisp. This is not the case, however, with high level library rules (see next section) used in conjunction with PRODOC. To help insure future generalizability of the PRODOC system, library rules fully reflect all of the constraints imposed on the rule construct as defined in the structural learning theory [7,22]. In that theory, the role of recursion is handled exclusively in terms of higher order rules (which may operate on other rules) and an universal control mechanism. Recursion is not allowed in individual rules. This restriction has been shown to have important implications in cognitive theory [19].

Scandura FLOWforms also are used to represent rule domain (input) and range (output) structures. In this context they are called “data FLOWforms”. In general, domain and range structures may be characterized mathematically as partial orderings. The various components/elements may be viewed as ordered sets whose elements in turn may be ordered sets.

<table>
<thead>
<tr>
<th>IF</th>
<th>THEN</th>
<th>ELSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE</td>
<td>DO</td>
<td>IF</td>
</tr>
</tbody>
</table>

Commands: Move keys, 1..9,f,a,b,r,Del,l,m,d,c,e,s,'s,z,g,l,w,?,?,F1,Esc

\[ \text{Domain A} \]

\[ \begin{array}{cccc}
    & & & \\
    & B & C & \\
    & E & F & H \\
\end{array} \]

In the structure below, set A has elements B and C; in turn, B is a (sub)set with elements E, F, and H; similarly, (sub)set C has element H. Although element H appears twice in this data FLOWform, it is simply a different display of the same element (something you can see when you edit one of them).

Although this representation looks similar to embedded CASE structures, the similarity is a bit deceptive. In procedures, CASE structures have both condition variables and operations. In data FLOWforms, all elements are the same; irrespective of level, any element may be refined into a set of elements.
Notice that data FLOWforms are not quite trees since element H belongs to both sets B and C. (A simple example of a tree is given later.)

Of course, partial orderings do include trees as a common subset.

Since data FLOWforms are restricted to partial orderings, it is true that they cannot directly represent cyclical relationships. Cyclic relationships are often used to summarize arbitrary connections among nodes (e.g., computer terminals) in a complex system. They also can be used to represent nonhierarchical data structures.

In the case of software development, however, this restriction is more apparent than real. In the former case, for example, the connections typically represent a sharing of data represented by the nodes. Just as data at any given node can be operated on by resident programs, programs also are needed to transfer data from one node to another. Thus, the cyclic networks themselves correspond to sets of programs, each of which may be represented in terms of a rule FLOWform. Such networks, in effect, provide a convenient way to represent the overall high level structure of a system of programs but they say relatively little about software development per se.

The following figure illustrates the latter case—data which a program procedure might operate on.
In this case, notice that the nodes "pillar 1" and "pillar 2" are superordinate to each other. This is not allowed in a partial ordering relationship. As with successive top-down structured refinement of procedures, most software engineers favor a hierarchical (partially ordered) approach to data structure design. Thus, for example, the above Arch structure might be represented hierarchically as:

where the definition of "supports" may include "not touching." In fact, the latter figure seems more natural. Accordingly, arches consist of two types of entity: supports and tops. In turn, (at least) two supports are needed.

The latter representation involves viewing the cyclic representation of the arch from a particular perspective, specifically where the pillars serve as "supports". The suppression of this role (support) in the cyclic relationship amounts to hiding critical information about the real world, and is to be avoided. More to the point, one can always construct a structured procedure equivalent to given "spaghetti" code. One, similarly, can always construct a partially ordered data structure, together with a procedure operating on it, which corresponds to any given cyclical data structure.

Cyclic relationships are useful for some purposes, as in relational data bases. From the perspective of a procedure operating on the data base, however, the cyclic data base is viewed as an unbounded (infinite) partial ordering (e.g., pillar1 → pillar2 → pillar1 → pillar2 → . . . ) for which the procedure must have a terminating condition. In any given application of such a procedure, a cycle is realized only a finite number of times. For example, the cyclic graph on the left (below) corresponds directly to a procedure executing a loop on the hierarchical graph on the right (with a stop condition on C.).

Using PRODOC, rule data structures and procedures are constructed in a top-down structured fashion and represented in terms of Scandura FLOWforms. As we have seen, FLOWforms look similar to Nassi-Shneiderman flow charts, but make better use of the rectangular screen and allow simultaneous display of as many (or as few) levels of representation as desired.

A procedure FLOWform having several levels of refinement might be displayed by PRODOC in any number of ways. In the illustration below all levels of a sort FLOWform are displayed. The highest level consists of a single high level description (component): "Sort up to 500 numbers; print result". Each higher level component, in turn, is decomposed into one or more lower level elements. At the next level, we have: "Specify the number of numbers to be sorted," "Prompt the user, then get the numbers," "Sort them," "Display description, then print the ordered set." (See Fig. 1.)
Sort up to 500 numbers; print result

- Specify the number of numbers to be sorted.

  write ('How many numbers (1 to 500) to be sorted? ')
  readln (n)

- Prompt user, then get the numbers.

  writeln ('Enter below numbers to be sorted. Press (Return) after each.')

  - Get the numbers.
  FOR i := 1 to n
  DO  readln (a[i])

  - Sort them.
  FOR i := 1 to n - 1
  DO  

       - Scan thru items and swap if necessary.
       
       FOR j := 1 to n - i
       DO  

            - Compare and swap if necessary.
            
            IF a[j] > a[j + 1]
            THEN  

                 - Swap

                 temp := a[j]
                 a[j] := a[j + 1]
                 a[j + 1] := temp

  - Display description, then print the ordered set.

  writeln

  writeln ('The resulting order is:')

  - Print the ordered set.
  FOR i := 1 to n
  DO  writeln (a[i]:2)
Given the ease with which FLOWforms may be read, PRODOC helps to break down communication barriers which frequently exist between end users (or clients), project managers and programmers. Suitable for use by nonprogrammers as well as programmers, PRODOC makes it possible to design, document, implement, and maintain software systems in an integrated, graphically supported, top-down structured environment. In addition to English text, the availability of greatly simplified, high level library rules also makes PRODOC useful in rapid prototyping. As we have seen, the availability of graphical support for input and output data structures also makes it possible to directly reflect arbitrary semantic properties.

All of this is accomplished within the context of four distinct but complementary and fully compatible software productivity and quality assurance environments:

(a) Applications Prototyping Environment (with English and library rule interpreter and expert assistant generator) (PRODOCea),
(b) Applications Prototyping Environment (for use with a Pascal compiler) (PRODOCip),
(c) Programming Productivity Environment (for use with any source code compiler or interpreter; supports Pascal, C, and Ada pseudo code) (PRODOCpp),
(d) Library Generator (for creating custom versions of PRODOCea) (PRODOClp).

Each of these environments (described below) makes use of Scandura FLOWforms. The first three also make use of a common and highly responsive visual interface for creating and editing FLOWforms interactively.

Relationships among the first three PRODOC environments as well as the way they may be used in developing applications software is represented schematically following (Fig. 2):

As shown in the figure, PRODOC may be used by domain experts to create FLOWform specifications in ordinary English. PRODOCea may be used to execute such specifications interpretively: In the sense that when PRODOCea comes to an operation it does not understand, it asks the human user to perform the step and enter the results obtained. In the case of decisions, PRODOCea asks the user to specify the condition satisfied before "executing" the appropriate next step.

The user may also enter high level library rules in FLOWform elements. When PRODOCea comes to a rule in its library, it executes that rule.
PRODOC Environment and Methodology

In fact, it is possible to mix library rules and English arbitrarily in constructing FLOWforms. The result in these cases is an “expert assistant” or performance aid available to guide and/or help perform the indicated task. Note: The term “expert assistant” has been used instead of “expert system” since PRODOCea does not currently include an inference engine, although there is no reason one could not be added. Expert assistants, however, are much easier to create using PRODOC. Assuming the creator knows exactly what the user is to do, useful expert assistants can be created in as little as five minutes time.

The current version of PRODOCea employs a fairly general but relatively low level set of library rules designed largely for testing purposes. The current library includes a variety of:

- input/output operations [e.g., display (ELEMENT, DISPLAY_PARAMETERS), load (DOS_NAME, DRIVE, FILE_TYPE)],
- other operations [(e.g., insert_component_after (VALUE, SET, PREVIOUS_COMPONENT), delete_component (SET, COMPONENT)],
- functions [e.g., add (ADDEND 1, ADDEND 2), modulo (X, BASE), find (VALUE, SET)],
- conditions [e.g., match (STRING 1, STRING 2), less_than (X,Y)],
- logical connectives [e.g., and (EXPRESSION 1, EXPRESSION 2)],
- and assignment (i.e., ELEMENT := VALUE).

The user also has the option of creating hierarchies of input/output data structures which directly reflect the reality they represent. Alternatively, inessential aspects of this structure may be suppressed. In this case, PRODOCea automatically generates a formal equivalent of the needed data structures (i.e., declarations). Once “initialized” in this way, PRODOC library rules may be executed immediately in interpretive mode for purposes ranging from simple execution to debugging.

In conjunction with PRODOC’s Library Generation facilities (see below), custom versions of PRODoca can easily be created, with rule libraries targeted at particular application areas. Customized in this way, PRODOCea makes it possible to represent arbitrary semantic properties, and facilitate rapid prototyping.

PRODOC1p is identical to PRODOCea in so far as prototype design and the use of library rules in rapid prototyping is concerned. Instead of an interpreter, however, PRODOC1p automatically generates Pascal source code from FLOWforms containing library rules. This source code may be compiled, linked with the PRODOC run time package and executed as an independent program. PRODOC1p’s generalized code generator also makes it possible to arbitrarily mix Pascal code with library rules, thereby gaining the prototyping advantages of any number of customized, arbitrarily high-level languages, along with the flexibility of Pascal. This feature makes it possible, for example, for a programmer to speed up or otherwise add finishing touches to a working prototype created by a nonprogrammer.

PRODOCcpp has all of the design, etc. features of the other PRODOC environments and is used where one wants to implement a design in some existing programming language (Pascal, C, Ada, COBOL, etc.).

In standard form, PRODOCcpp supports source code in any programming language. (Incidentally, PRODOC can be used as a full-function idea processor. This text, for example, was prepared using PRODOC exclusively.)

In addition, pseudo code support is available as a current option for Pascal, C and Ada. In this case, the user enters pseudo code into FLOWform displays. Syntax and other routine aspects of code generation (e.g., BEGINS..END, etc.) are all generated automatically. The result effectively combines the clarity and ease of use of high-level fourth generation languages with the flexibility of third generation languages. These options include syntax checking, consistency checking and automatic declarations generation. Also underway is an utility which will automatically convert existing source code into FLOWforms containing corresponding pseudo code and design information. Support for other third and fourth generation languages will be added as needed.

A sample FLOWform for sorting numbers and the corresponding Pascal source code are shown in Figure 3.
Sort up to 500 numbers; print result

```
write ('How many numbers (1 to 500) to be sorted? ')
readln (n)
writeln ('Enter below numbers to be sorted. Press <Return> after each. ')
FOR i := 1 to n
  DO readln (a[i])
FOR i := 1 to n - 1
  DO FOR j := 1 to n - i
     DO IF a[j] > a[j + 1]
        THEN temp := a[j]
             a[j] := a[j + 1]
             a[j + 1] := temp
     END
  END
writeln
writeln ('The resulting order is: ')
FOR i := 1 to n
  DO writeln (a[i]:2)
```

**Fig. 3.**

```
PROGRAM sort;
    VAR n : INTEGER;
    i : INTEGER;
    a : ARRAY[1..500] OF INTEGER;
    j : INTEGER;
    temp : INTEGER;
BEGIN
  { Sort up to 500 numbers; print result }
  BEGIN
    write ('How many numbers (1 to 500) to be sorted? ');
    readln (n);
    { Prompt user, then get numbers. }
    BEGIN
      write ('Enter below numbers to be sorted. Press (Return) after each. ');
    END
  END
  FOR i := 1 to n
    DO readln (a[i])
  END;
  { Get the numbers from the user. }
  FOR i := 1 to n
    DO FOR j := 1 to n - i
       DO IF a[j] > a[j + 1]
          THEN BEGIN
             temp := a[j];
             a[j] := a[j + 1];
             a[j + 1] := temp
          END
       END
  END;
  { Sort them. }
  FOR i := 1 to n - 1
    DO BEGIN
       { Scan thru items and swap if necessary. }
       FOR j := 1 to n - i
         DO BEGIN
            { Compare and swap if necessary. }
            IF a[j] > a[j + 1]
               THEN BEGIN
                  temp := a[j];
                  a[j] := a[j + 1];
                  a[j + 1] := temp
               END
         END
    END
END;
```
PRODOC Environment and Methodology

BEGIN
writeln;
writeln('The resulting order is:');
{ Print the result. }
FOR i = 1 to n DO
writeln(a[i]:2)
END
END

PRODOC C also makes it possible to reverse the process. Existing system design and specification data, as well as source code, may automatically be loaded into FLOWform elements. From there, it may be broken down successively using PRODOC until each element contains a single pseudo code statement. A PRODOC utility to assist in the process is available in the case of Pascal. Current plans call for extending support to Ada and other languages as needed. In turn, given this pseudo code, PRODOC C will almost instantaneously generate full source code in Pascal, C, or Ada [23,24].

It should be emphasized that the use of PRODOC is not limited to the development of small systems. PRODOC was developed (after some early “bootstrapping”) and currently is fully maintained in terms of FLOWforms using PRODOC (under MS-DOS). PRODOC consists of about a quarter million lines of code. Intelligent Micro Systems, Inc.’s ability to develop such a system with limited resources in so short a period of time is due in no small part to PRODOC’s ability to produce complex systems that are robust, reliable and easy to change. Moreover, we cannot foresee any intrinsic limits on the size of the systems PRODOC can be used to create. This will be especially true after a planned port to Unix has been completed.

The fourth PRODOC environment (PRODC C) makes it possible to create new libraries, targeted at particular families of applications, from (portions of) existing rule libraries and/or other library rules. Targeted libraries may be integrated with the basic PRODC C system, thereby creating any number of customized versions of PRODC C. Since doing so requires access to PRODOC source code, creating customized versions of PRODC C will normally involve a collaborative effort between Intelligent Micro Systems, Inc. and software specialists in particular application areas.

The use of PRODC C in developing customized versions of PRODC C is represented schematically in Figure 4.

SUMMARY OF THE SYSTEM DEVELOPMENT METHODOLOGY

Collectively, the various PRODOC environments provide a fully integrated and comprehensive software development system including: requirements definition, systems design and documentation, prototype development, code generation and program maintenance. Rules (represented in terms of data structure and procedure FLOWforms) provide an unique visual and uniform representation that can be used throughout.

The PRODC C applications prototyping environment is designed primarily for use by system designers and/or application experts (in conjunction with intended users). Given a reasonable degree of computer awareness, PRODC C also can and has been used independently by end users.

In this context, PRODC C can be used in system analysis and requirements definition. System analysis normally involves very high level descriptions of the various system states (data structures) and processes in ordinary English. Data FLOWforms will normally be used to describe the states, and transitions between states will be described at a high level in terms of procedure FLOWforms. Should the designer wish, these process de-
PRODOC Environment and Methodology

scripts may include hardware, personnel and other
development requirements.

During the requirements definition phase,
users will develop more detailed descriptions of the
key states and transitions. This is accomplished by
successive refinement of the very high-level system
descriptions, all in an integrated FLOWform
environment.

Representing system analysis and require-
ments definition in terms of data and procedure
FLOWforms has another useful feature. PRODOCea
makes it possible to "execute" systems analyses and/or
requirement definitions dynamically. That is, one
can simulate transitions between various states of
the to-be-developed system, thereby giving the user
some feeling for how the system might operate in
practice.

As is well known, the distinction between re-
quirements definition and program design is largely
arbitrary and depends on one's perspective. In the
former case, definition of the key states of the system,
and of the transition procedures connecting them are
described in largely functional, real world terms.
Conversely, program designs typically are repres-
tented in terms of constructs associated with parti-
cular programming languages. (In this context, a ma-
jor advantage of the Ada programming language is
its relative flexibility, thereby making it possible to
more closely reflect system requirements in design
specifications [1]. A concomitant disadvantage of Ada
is its relative complexity, one which the use of PRO-
DOCcpp with its Ada pseudo code support helps to
alleviate.)

Unlike the other PRODOC environments,
PRODOCea prototyping environments are associ-
ated with given atomic rule libraries. Since rule li-
braries are designed to accommodate particular fam-
ilies of applications, both the data structures these
rules operate on, as well as the rules themselves,
directly reflect application realities. Consequently,
where a program specification is represented in terms
of a FLOWform in which the terminal elements are
all library rules, this specification may be inter-
preted directly (by PRODOCea).

It may be possible to directly create an oper-
ational system even where the terminal elements of
a program specification are not yet available as li-
brary rules. This might be accomplished in either of
two ways:

One, new atomic library rules might be se-
lected (or constructed) from available libraries and/or
created anew (e.g., using PRODOCcpp). These new
library rules can be integrated automatically to form
a new library using PRODOCig. PRODOCig gen-
erates complete Pascal code which can, in turn, be
linked with PRODOCea to create a customized pro-
totyping environment. The new PRODOCea, then,
can be used to directly interpret the original program
specification or prototype (formulated in terms of
atomic rules in the new library).

PRODOCig serves a supplemental role in this
context. Although the prototype can be interpreted,
tested and debugged using PRODOCea, execution
efficiency can usually be greatly improved via com-
pilation. In this case, PRODOCig can be used to con-
vert the given prototype (represented solely in terms
of library rules and meaningful data structures) into
Pascal source code ready for compilation.

Pascal pseudo code also can be used to sup-
plement library rules in a given prototype. This can
be done without restriction. Given the resulting li-
brary/Pascal pseudo code combination, PRODOCig
again can be used to generate complete Pascal source
code.

Two, the existing program specification (e.g.,
requirements definition) might be further refined as
normally is done into a detailed system design. In
this case the data structures and procedures (rep-
resented in terms of applications reality) are refor-
mulated in terms of data structures and operations
more closely associated with some target source lan-
guage. These more detailed designs, then, are con-
tered to code using PRODOCcpp. For this purpose,
one can enter complete source code in any pro-
gramming language using PRODOCcpp's default "text"
mode. Alternatively, one can simply enter Pascal, C,
or Ada pseudo code. In this case, syntax and consis-
tency checking, as well as declarations and source
code generation, are performed automatically.

Strict adherence to the foregoing methodology
has important benefits in program maintenance.
Given the integrated, fully interchangeable nature of
the various PRODOC environments, there is no justifiable reason why system requirements, design,
program documentation or source code should ever
get out of synchronization. Consequently, "finding
one's way around" even in very complex systems is
several orders of magnitude easier than is normally
the case.

In developing smaller programs, of course, it
may be possible to bypass some of the above steps.
Thus, one has the choice of creating and simply using
an applications prototype as is, or of designing and
coding the program using PRODOCcpp directly (e.g.,
in conjunction with particular sets of PRODOCcpp
pseudo code language support files).

At this point, it may be unclear how we pro-
pose to deal with the various other representational
systems that are commonly used by designers. In this regard, we take essentially the same position that Martin and McClure [3] take with respect to their "action diagrams": Although the methodologies may appear to differ, all of the commonly used forms of representation are either equivalent (to ours) or incomplete. In fact, while action diagrams are similar in some respects to FLOWforms, they do not incorporate design or display overall structure nearly as clearly.

Overall, PRODOC's single most important advantage is breaking down communication barriers between end users (or clients), project managers, designers, implementation programmers, documenters and maintenance staff. (This is equally true where single individuals serve multiple roles over a period of time.) Using PRODOC has the advantage of placing requirements definition, systems design, prototyping, and program coding (not to mention system maintenance) on the same plane. System designs, prototypes, and program code are viewed within an integrated environment, which is far easier to understand, revise, debug, and modify than is normally the case. Developing and maintaining executable (interpretable or compilable) prototypes and/or source code is a natural extension of system design and documentation, and vice versa.

Those of use who have been involved in the creation of PRODOC are fond of pointing out that PRODOC has literally been indispensable in its own creation. Indeed, we would not even consider taking on a new programming task without it.

REFERENCES

23. R. Hooi, and J. Giarratano, "Software issues involved in code translation of C to Ada programs," In proceedings First international conference on Ada programming language applications for the NASA space station (vol. 1), Clear Lake, TX (June 2–5, 1986).