Application and Systems Software in Ada: Development Experiences

Jim Kuschill
Computer Representatives, Inc.
Santa Clara, California

This presentation focuses on two issues: why CRI chose to convert its existing commercial software products to Ada and the the technical challenges we faced both before and during the rewrite process. The presentation will cover the following:

I. Environment
   A. Began the rewrite of software written in SPL and FORTRAN to Ada in 1983.
      1. Software included: relational DBMS, 4GL tools, and project management system.

II. Why Ada?
   A. Current and future maintenance considerations.
   B. Transportability had tremendous marketing advantages.

III. Planning Challenges
   A. Shortage of available programmers.
   B. Learning curve amongst own personnel.
   C. Unknown degree of diffulculty in the use of Ada for the development of application software.

IV. Technical Challenges
   A. Strong typing requirements of Ada affected the data conversions necessary for relational accessing.
   B. Ada packaging functions forced some new coding and routines to be written for an already mature product.
   C. Overloading capability smoothed the transition between some functions.

V. Opinions and Results
   A. The re-write process totaling approximately 250,000 lines of code is now in alpha test (will be in beta by the time of the SIGAda Conference).
   B. The learning curve was shorted than expected and differed by the nature of the language each programmer was accustomed to using previously.
   C. Maintenance problems and costs, as demonstrated during development will be vastly reduced as a result of Ada.
   D. The structure of Ada forces the writing of better routines, therefore better software.
   E. The time between a successfully compiled program and a completed program is drastically reduced because of Ada strict coding requirements.

N89-16370

Software Development: The PRODOC Environment and Associated Methodology

Joseph M. Scandura, Ph. D.
University of Pennsylvania

In its most basic sense software development involves describing the tasks to be solved -- including the given objects and the operations to be performed on those objects. Moreover, such descriptions must be precise in order for a computer (or human) to perform as desired. Unfortunately, the way people describe objects and operations typically bears little resemblance to source code in most contemporary computer languages.

There are two potential ways around this problem. One is to allow users to describe what they want the computer to do in everyday, typically imprecise English (or to choose from a necessarily limited menu of choices). This approach has some obvious advantages and a considerable amount of research is underway in the area. The approach, however, also has some very significant limitations: (a) it currently is impossible to deal with unrestricted English, and this situation is unlikely to change in the foreseeable future; and (b) even if the foregoing limitation is eventually overcome, the approach would still require the addition of complex, memory intensive "front ends". These "front ends" interact with the user's typically imprecise English statements and effectively "try to figure out" what the user intends. The result invariably is a system which is both sluggish in performance and limited in applicability.

The PRODOC methodology and software development environment is based on a second, we believe sounder, more flexible and possibly even easier to use approach. Rather than "hiding" program structure, PRODOC represents such structure graphically using visual programming techniques. In addition, the program terminology used in PRODOC 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 might easily be learned by an intelligent human (in a few minutes time). The approach taken with PRODOC is general, as well as efficient and easy to use.

PRODOC employs a unique graphically supported approach to software development, and supports the entire systems software development process, from requirements definition and system design to prototyping, code generation and maintenance. Although radically different at a superficial level, PRODOC draws generally on our extensive research in structural learning (the science of cognitive, instructional and intelligent systems engineering, Scandura, 1986). It represents a major step in the direction of automating the process of Structural (cognitive task) Analysis (e.g., Scandura, Durnin & Wulfeck, 1974; Scandura, 1977, 1982, 1984a, 1984b). More specifically, a special rule construct (not to be confused with production rules) plays a particularly central role in PRODOC.

C-6

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

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 (e.g., Scandura, 1970). Rules have been shown to provide a convenient way to represent a wide variety of human cognitive processes as well as arbitrary computer systems (e.g., Heller & Reif, 1984; Scandura, 1969, 1971, 1973, 1977; Scandura & Scandura, 1980).

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 schemes (i.e., classes of problems) and refer to input, output and intermediate (local) structures. Collectively, they correspond to the data structures on which programs operate. These correspondences are summarized below:

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

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 nondivisible for present purposes. 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." These include PRODOC "library rules". The extended version of PRODOC makes it possible to create libraries of such rules. These libraries make it easy for nonprogrammers (as well as programmers) to construct executable PRODOC rules.

As mentioned above, rules may be written in a language which is either understandable to humans and/or interpretable by computer. In either case, however, the same basic form of representation may be used. FLOWforms are easily understood by most people and can be used to represent arbitrary procedures (whether rule procedures or program procedures).

Like all structured procedures, FLOWforms may be refined arbitrarily. They are used for two purposes, one to represent procedures and two, to represent input/output data structures.

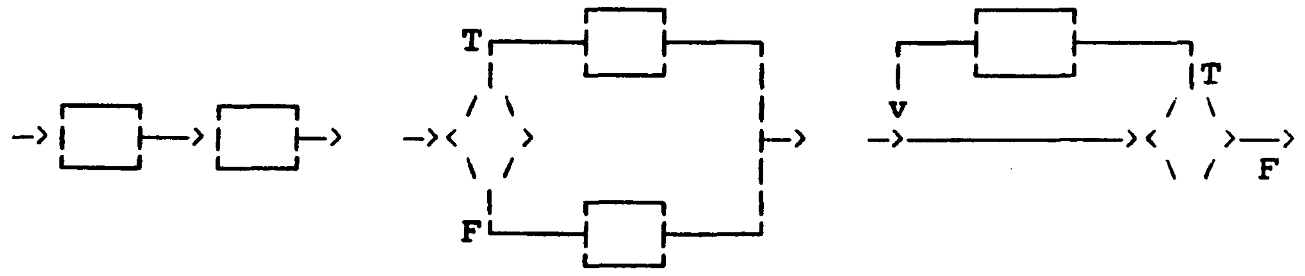
Roughly speaking, a procedure or algorithm is a recipe, process, technique, or systematic method for doing something. (The term "algorithm" is often preferred in computer science.) More precisely, according to Knuth (1968), a procedure or algorithm must:

(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 insure this property, each step in a structured procedure must be decomposable into one of three basic types of components;

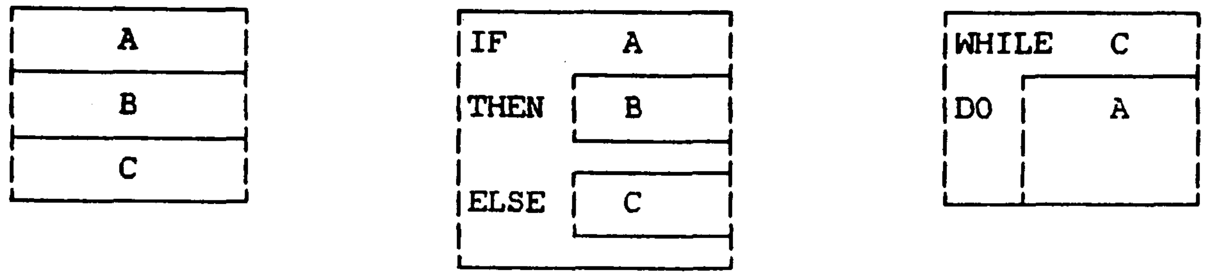
(a) Sequence of steps or operations,
(b) Conditional steps or branching (selection) and
(c) Iteration or looping.

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...).



(a) (b) (c)

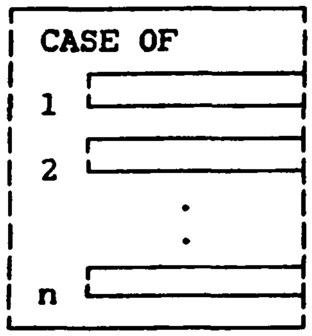
In Scandura FLOWform these three types of components are represented as shown below.



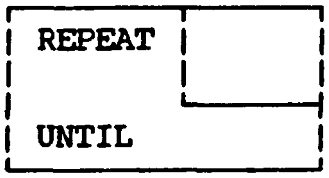
Sequence Selection (IF..THEN..ELSE) Iteration (WHILE..DO)

These three basic types of decomposition are univerally 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.

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

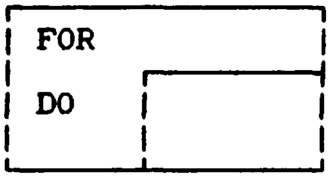


Selection (CASE)



Iteration

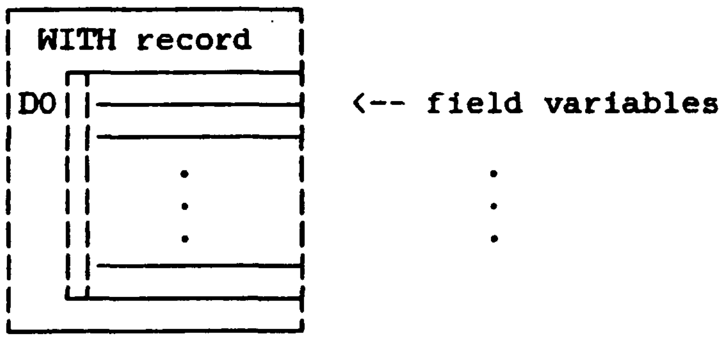
(REPEAT...UNTIL)



Iteration

(FOR...DO)

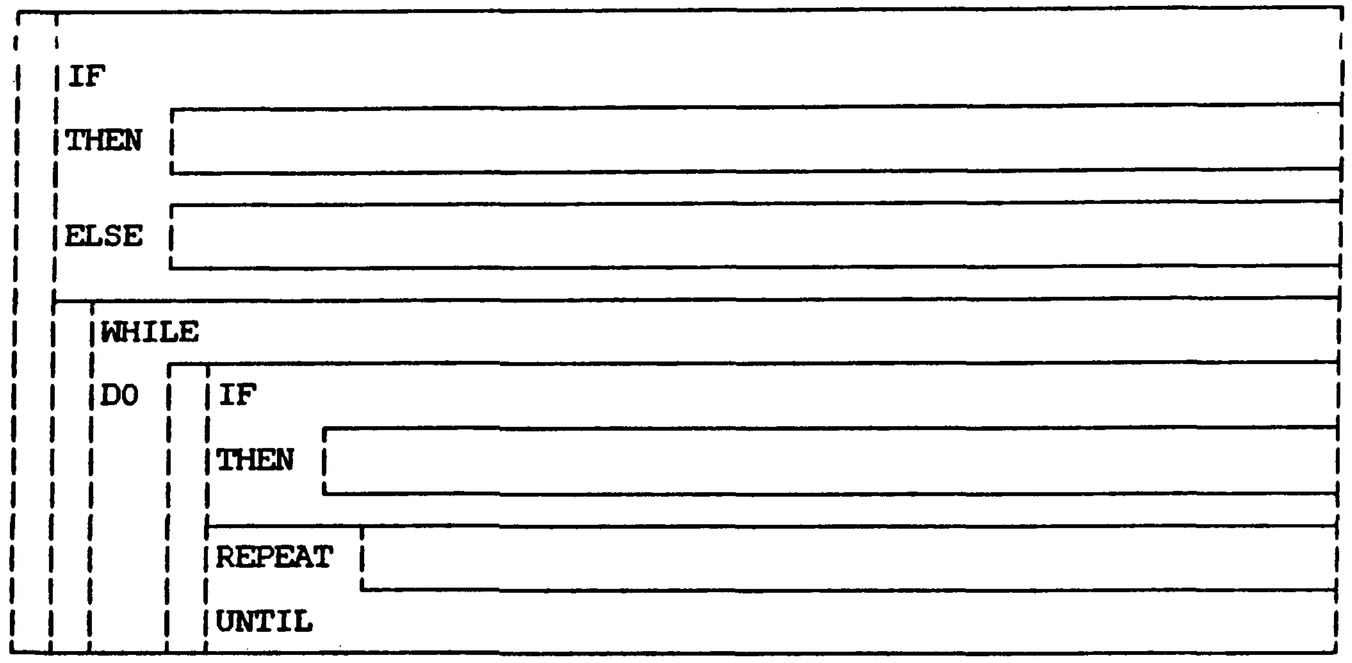
Although it does not fall into one of the three basic classes, Pascal also supports a WITH (Record..Do) structure. This is represented in FLOWforms as:



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:

```
[SAMPLE_R]:sample_FLOWform_structures     Copyright 1986  Scandura
```



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

Parenthetically it is worth noting that FLOWform procedures 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 (e.g., Scandura, 1977, 1981). 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 for diagnostic testing and learning (e.g., Scandura, 1980.)

Scandura FLOWforms also are used to represent rule domain (input) and range (output) structures. 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.

```
             Domain A
               /\
              /   \
            B       C
          / | \    /|
         /  |  \  / |
        /   |   \   |
      E     |   /  \|
            F  G    H
```

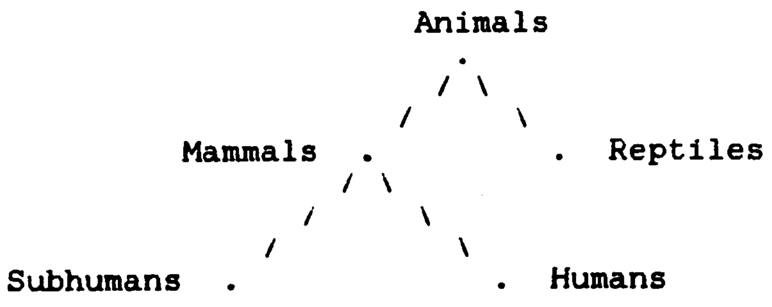
In the structure below, set A has elements B and C; B has elements E, F and H; C has G and H. Although element H appears twice in this FLOWform, it is simply a different display of the same element (something you will see when you edit one of them).

Although this representation looks similar to the CASE structure, the similarity is a bit deceptive. In procedures, CASE structures have both condition variables and operations. The condition occupies a distinguishing position to the right of the word "CASE" and may be thought of as the first CASE element.

```
[SAMPLE]:Sample_DOMAIN_FLOWform             Copyright 1986  Scandura

┌──────────────────────────────────────────────────────────────────┐
|                                                                  |
|[DOMAIN]:                                                         |
| ┌────────────────────────────────────────────────────────────────┤
| |[A]:                                                            |
| | ┌──────────────────────────────────────────────────────────────┤
| | |[B]:                                                          |
| | | ┌────────────────────────────────────────────────────────────┤
| | | |[E]:                                                        |
| | | └────────────────────────────────────────────────────────────┤
| | | ┌────────────────────────────────────────────────────────────┤
| | | |[F]:                                                        |
| | | └────────────────────────────────────────────────────────────┤
| | | ┌────────────────────────────────────────────────────────────┤
| | | |[H]:                                                        |
| | | └────────────────────────────────────────────────────────────┤
| | └──────────────────────────────────────────────────────────────┤
| | ┌──────────────────────────────────────────────────────────────┤
| | |[C]:                                                          |
| | | ┌────────────────────────────────────────────────────────────┤
| | | |[H]:                                                        |
| | | └────────────────────────────────────────────────────────────┤
Commands:Move keys,1..9,f,a,b,r,Del,t,m,d,c,e,s,^,z,g,l,w,?,F1,Esc
```

Notice that this representation is not quite a tree since element H belongs to both sets B and C. Of course, partial orderings do include trees as a common subset. A simple example of a tree is given below.



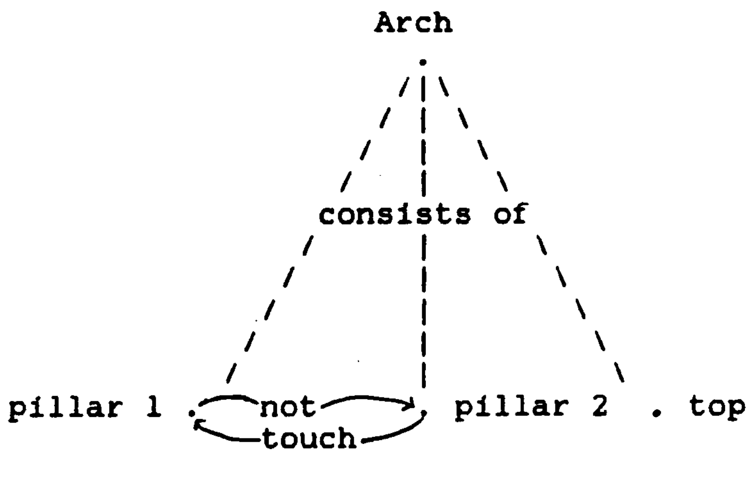
Since rule data structures are restricted to partial orderings it is true that FLOWforms cannot directly represent cyclical relationships. In the case of software development, however, this restriction is more apparent than real. Cyclic relationships can serve two quite different purposes:

(1) They can be used to summarize connections among nodes (e.g., computer terminals) in a complex system.

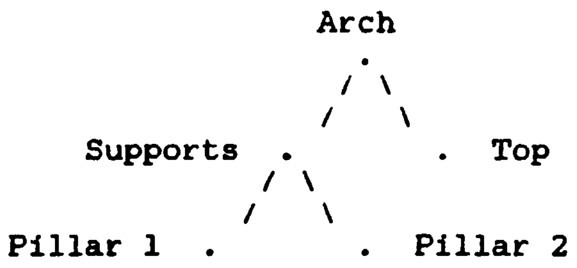
(2) They can be used to represent nonhierarchical data structures, where the relationships are not necessarily monotonic.

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 figure below illustrates the latter case -- data which a program procedure might operate on.



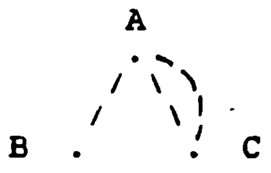
In this case, notice that the nodes "pillar 1" and "pillar 2" are superordinate to each other. This is not allowed in a partial ordering relationship. As with successive top-down structured refinement of procedures, most software engineers favor a hierarchical (partially ordered) approach to data structure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

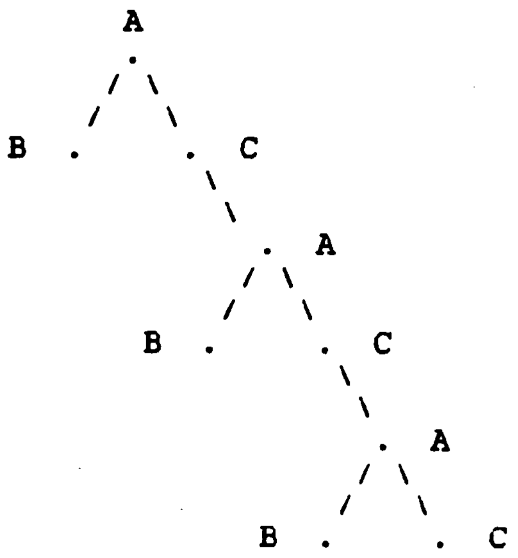


where the definition of "supports" may include "not touching". In fact, the latter figure seems more natural. Accordingly, arches consist of two types of entity: supports and tops. In turn, (at least) two supports are needed.

Nonetheless, it is fair to ask whether cyclic relationships are necessary for some purposes. While we do not know of any definitive answer to this question, it would appear that the answer is "no". Just as any procedure can be represented as a structured procedure, cyclic data structures can be represented in terms of partial orderings. To see this, notice that cycles correspond to infinite hierarchies (e.g, pillar 1 --> pillar 2 --> pillar 1 --> pillar 2 -->).

However, any given cycle can be realized only a finite number of times in the real world. Hence, cyclical relationships can be represented by finite successive refinement of the cycles in question. Consider, for example, the cyclic graph on the left (below) and the equivalent partial ordering on the right. While the cyclic graph looks simpler, it camouflages the fact that the cycle is repeated only twice.





In effect, the apparent loss of representational simplicity is at least partially overcome by the more precise characterization provided by the partial ordering. The suppression of such details is not appropriate in actual software development.

It would appear, just as one can always construct a structured procedure equivalent to given "spagetti" code, one can always construct a partially ordered data structure that is equivalent to any given cyclical data structure.

PRODOC

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 they make better use of the rectangular screen and allow simultaneous display of as many (or as few) levels of representation as may be desired.

A procedure FLOWform with several levels of refinement might be displayed by PRODOC as illustrated below. At the highest level, for example, data structures and procedures each consist of a single high level description (component). Various components, in turn, are decomposed into one or more lower level elements.

Insert FLOWform showing several levels

PRODOC consists of four distinct but complementary and fully compatible software productivity and quality assurance environments. Each of these environments (described below) makes use of Scandura FLOWforms.

Relationships among the first three PRODOC environments as well as the way they may be used in developing applications software is represented schematically on the following page.

Insert first schematic here
----

(1) Applications Prototyping Environment (with interpreter and expert assistant generator) (PRODOCea) - is suitable for use by nonprogrammers as well as programmers for designing, documenting, implementing, and maintaining 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 makes PRODOCea ideal for rapid prototyping. The availability of graphical support for input and output data structures also makes it possible to directly reflect arbitrary semantic properties.

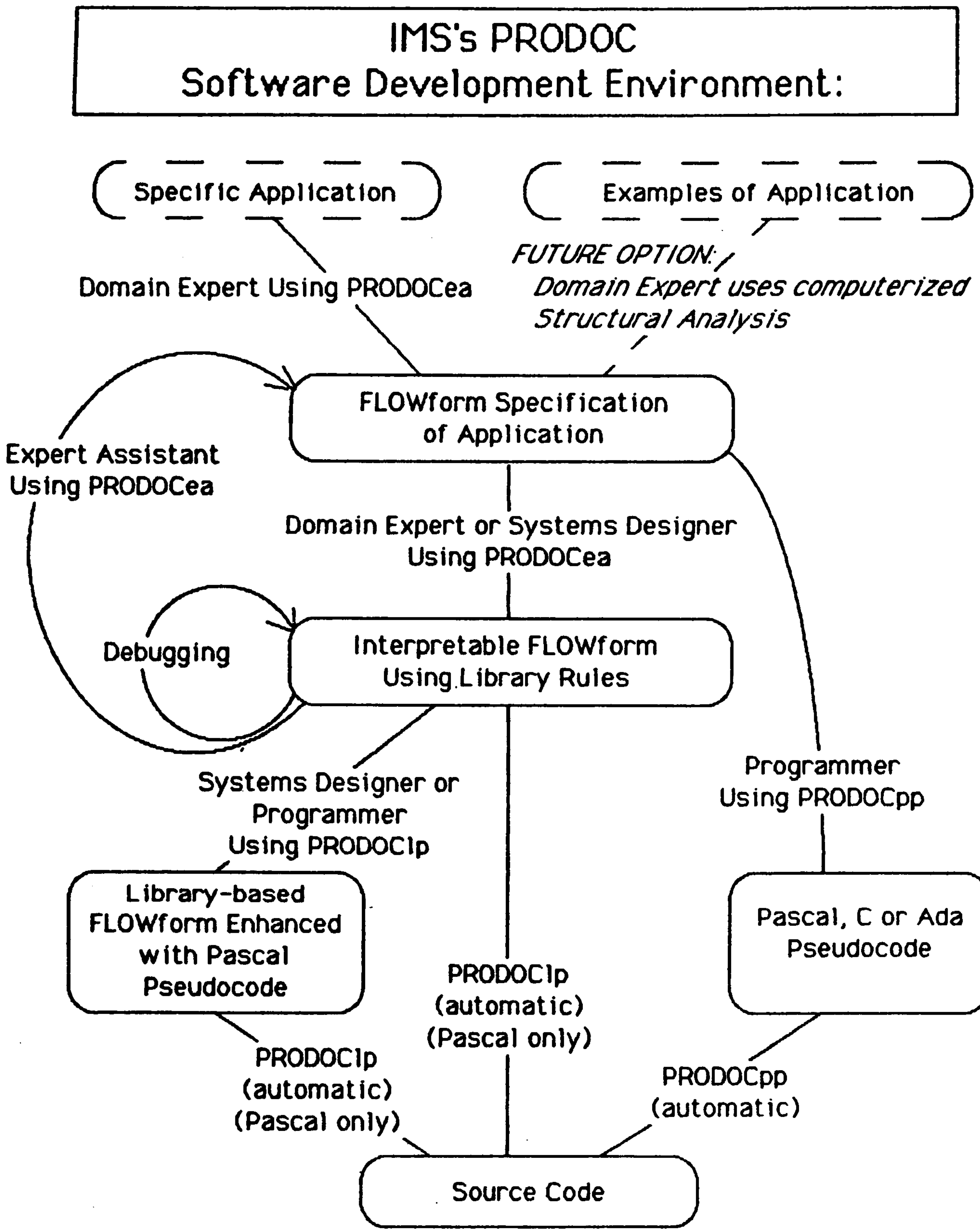
The current version of PRODOCea employs a fairly general but relatively low level set of library rules designed largely for testing purposes. The current library includes a variety of:

input/output operations [e.g., display (ELEMENT, 05-12-86

[SORT]:sort

Sort up to 500 numbers;print result

```
write ('How many numbers (1 to 500) to be sorted?  ')
readln (n)
. Prompt user, then get numbers.
    writeln ('Enter below numbers to be sorted. Press <Return> after each.')
    . Get the numbers from the user.
    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
. Identify and then print the resulting ordered set.
    writeln
    writeln ('The resulting order is:')
    . Print the result.
    FOR i:= 1 to n
    DO  writeln (a[i]:2)
```


IMS's PRODOC
Software Development Environment:
Specific Application
Examples of Application
FUTURE OPTION:
Domain Expert uses computerized Structural Analysis
Domain Expert Using PRODOCea
FLOWform Specification of Application
Expert Assistant Using PRODOCea
Domain Expert or Systems Designer Using PRODOCea
Debugging
Interpretable FLOWform Using Library Rules
Systems Designer or Programmer Using PRODOClp
Programmer Using PRODOCpp
Library-based FLOWform Enhanced with Pascal Pseudocode
PRODOClp (automatic) (Pascal only)
Pascal, C or Ada Pseudocode
PRODOClp (automatic) (Pascal only)
PRODOCpp (automatic)
Source Code

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, PRODOC 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 (4) below), custom versions of PRODOCea (and PRODOClp) can quickly be created to accommodate library rules to facilitate rapid prototyping in arbitrary semantic properties.

A unique feature of PRODOCea is its ability to immediately execute not only interpretable library rules but statements written in ordinary English. This makes it possible to actually run through a proposed system design before it has even been prototyped in terms of high level library routines, let alone reduced to standard program code. An additional advantage is that it makes the difficult and expensive process of developing many expert systems almost trivial. Once an (nonprogrammer) expert knows what a human/computer assistant is to do, it is a simple task to develop a computerized expert assistant or performance aid to assist less qualified personnel in performing the required tasks.

(2) Applications Prototyping Environment (for use with a Pascal compiler) (PRODOClp) - 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, PRODOClp includes a much generalized code generator which 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.

(3) Programming Productivity Environment (PRODOCpp) - has all of the design, etc. features of PRODOCea. PRODOCpp comes in standard form which 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 an option for Pascal, C, Ada and other programming languages. For example, Pascal, C and Ada syntax and other routine aspects of code generation (e.g., BEGINS..ENDS, 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 also include syntax checking, consistency checking and automatic declarations generation. Current plays call for adding pseudo code support for other third and fourth generation languages as needed.

A sample FLOWform for sorting numbers and the corresponding Pascal source code are shown on the next page.

Insert Sort FLOWform and Code

----

(4) Library Generator (PRODOClg) - makes it possible to integrate available rule libraries and new library rules into either PRODOC prototyping environment, thereby creating customized versions of PRODOC for particular families of applications. Since this requires access to PRODOC source code, customized versions of PRODOC will normally involve a collaborative effort involving our development team and software specialists in particular application areas.

The use of PRODOClg in developing customized versions of PRODOCea and PRODOClp is represented schematically on the next page.

Insert schematic for PRODOClg here

01-23-86

[SORT]:sort

Sort up to 500 numbers;print result

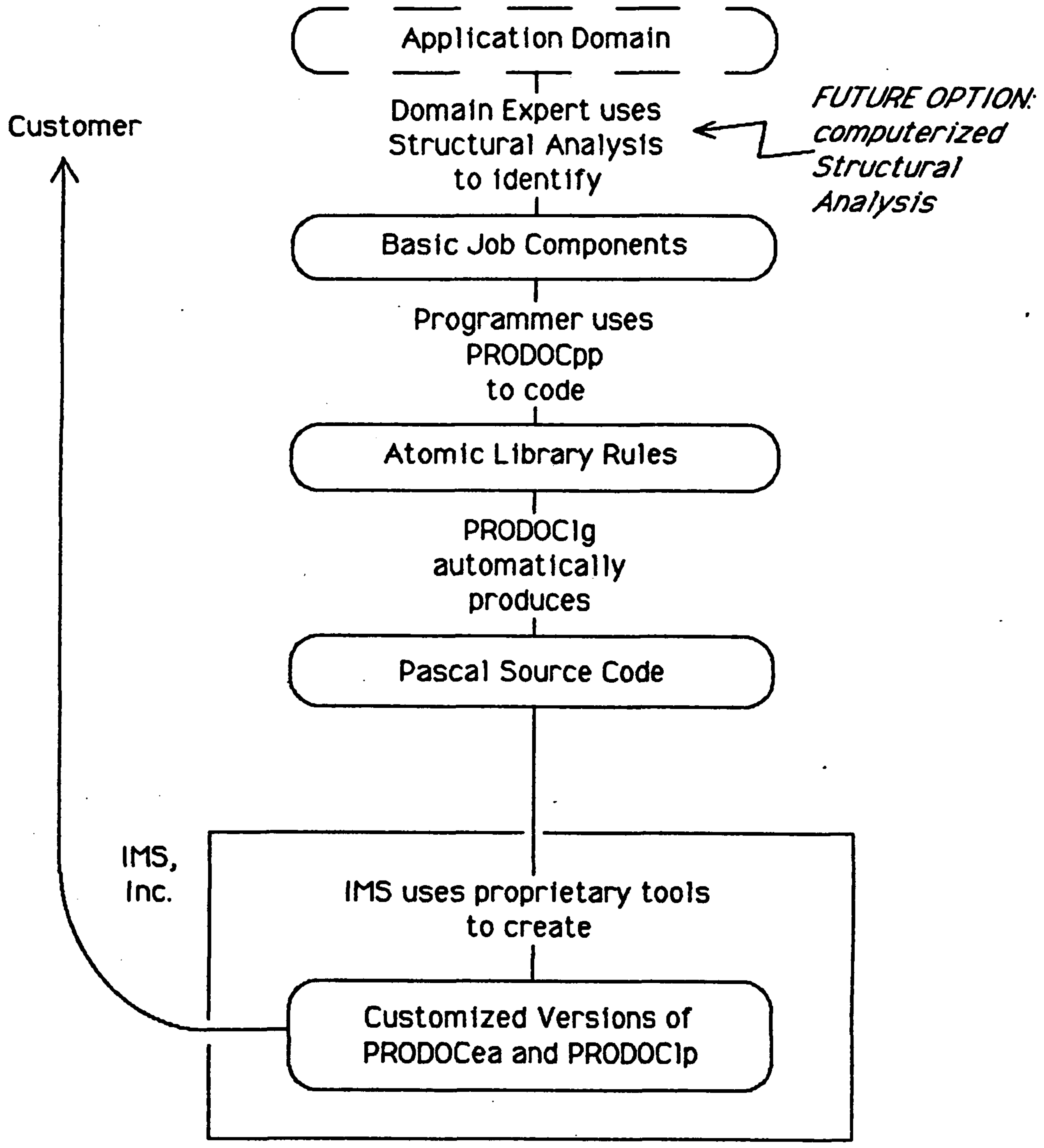
```
write ('How many numbers (1 to 500) to be sorted?  ')
readln (n)
    writeln ('Enter below numbers to be sorted. Press <Return> after each.')
    FOR i:=1 to n
    DO  readln (a[i])
FOR i:= 1 to n-1
DO  FOR j:= 1 to n -i
    DO  IF a[j]> a[j+1]
        THEN  temp:= a[j]
              a[j]:= a[j+1]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

```
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
      writeln ('Enter below numbers to be sorted. Press <Return> after each.');
      { Get the numbers from the user. }
      FOR i:=1 to n DO
        readln (a[i])
    END;
    { 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. }
        BEGIN
        IF a[j]> a[j+1] THEN
          { Swap }
          BEGIN
            temp:= a[j];
            a[j]:= a[j+1];
            a[j+1]:= temp
          END
        END;
    { Identify and then print the resulting ordered set. }
    BEGIN
      writeln;
      writeln ('The resulting order is:');
      { Print the result. }
      FOR i:= 1 to n DO
        writeln (a[i]:2)
    END
  END
END.
```

IMS's Structural Analysis Methodology and PRODOClg Library Generator:


Application Domain
Domain Expert uses Structural Analysis to identify
FUTURE OPTION: computerized Structural Analysis
Customer
Basic Job Components
Programmer uses PRODOCpp to code
Atomic Library Rules
PRODOClg automatically produces
Pascal Source Code
IMS, Inc.
IMS uses proprietary tools to create
Customized Versions of PRODOCea and PRODOClp

OVERVIEW OF THE SYSTEM DEVELOPMENT METHODOLOGY

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

The PRODOCea applications prototyping environment is designed primarily for use by system designers (in conjunction with intended users). (Given some initial training, in fact, it also can and has been used independently by end users.)

In this context, PRODOCea can be used in system analysis and requirements definition. System analyses will normally involve 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 descriptions 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 environment.

PRODOCea makes it possible to "execute" these 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 a better feeling for how the system might operate in practice.

As is well known, the distinction between requirements 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 represented in terms of constructs associated with programming languages.

The various PRODOC prototyping environments are associated with given atomic rule libraries. Since rule libraries are designed to accommodate particular families of applications, both the data structures these rules operate on, as well as the rules themselves, directly reflect application realities.

Consequently, library rules (including both data structures and atomic rules) might be used directly in the case of requirements definition. Indeed, the resulting definitions might be interpreted directly (by PRODOCea) where the terminal (most refined) elements of the key transition procedures correspond to atomic rules in the associated library.

It may, in fact, still be possible to directly create an operational system even where the terminal elements of a systems definition or design are not already available as library rules. This might be accomplished in either of two ways:

(1) New library rules might be selected from available libraries and/or created (e.g., using PRODOCpp). These new rules can be integrated automatically to form a new library using PRODOClg. PRODOClg generates complete Pascal code which can, in turn, be linked with either PRODOC prototyping environment to create a custom version (of either). This new custom version, then, can be used to directly interpret the original systems definition or design (formulated in terms of atomic rules in the new library).

(2) The requirements definition stage might be further developed as normally is done into a detailed system design. In this case the data structures and procedures (represented in terms of applications reality) are reformulated in terms of data structures and operations more closely associated with some target source language. These more detailed designs, then, are converted to code using PRODOCpp. For this purpose, one can enter complete source code using PRODOCpp's default "text" files. Alternatively, in conjunction with available language-specific files, one can simply enter pseudo code. In the latter case, syntax and consistency checking and declarations and source code generation, may be performed automatically.

PRODOClp serves a supplemental role in the above context. For example, Pascal pseudo code can be used to supplement whatever library rules happen to be used in a given design. This can be done without restriction. Given the resulting library/Pascal pseudo code combination, PRODOClp can be used to generate complete Pascal source code ready for compilation.

PRODOClp also serves a useful role even where all elements of a design consist of library rules. Although the design can be interpreted, tested and debugged using PRODOCea, execution efficiency can usually be greatly improved via compilation. In this case, PRODOClp can be used to convert the given design (represented solely in terms of library rules and meaningful data structures) into complete Pascal source code ready for compilation.

Perhaps the single most important advantage in following the foregoing methodology is that of program maintenance. Given the integrated, fully interchangeable nature of the various PRODOC environments, there is no justifiable reason why system requirements or design, program documentation, or code should ever get out of synchronization. Consequently, finding one's way around in even very complex systems is several orders of magnitude easier than is normally the case. Furthermore, the printed documentation provides additional features that are especially useful with large system segments.

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 PRODOCpp directly (e.g., in conjunction with particular sets of PRODOCpp pseudo code language support files).

At this point, it may be unclear how we propose 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 (1985) 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 formally equivalent to procedure FLOWforms, we do not believe that they display overall structure nearly as clearly.

By way of summary, 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. Put somewhat differently, developing and maintaining executable (interpretable or compilable) prototypes and/or source code is a natural extension of system design and documentation, and vice versa. In short, PRODOC supports the entire systems software management and development process, from requirements definition to code generation.

Those of us 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 using PRODOC.

REFERENCES

Heller, J. I. and Reif, F. Prescribing effective human problem-solving processes: Problem description in physics. Cognition and Instruction, 1, 177-216, 1984.

Knuth, D. E. The Art of Computer Programming Vol. 1: Fundamental Algorithmes. Reading, MA: Addison-Wesley, 1968.

Martin, J. and Mc Clure, C. Action Diagrams: Clearly Structured Program Design. Englewood Cliffs, NJ: Prentice-Hall, 1985.

Scandura, J. M. New directions for theory and research on rule learning: II. empirical research, Acta Psychologica, 1969, 29, 101-133.

Scandura, J. M. The role of rules in behavior: toward an operational definition of what (rule) is learned. Psychological Review, 1970, 77, 516-533.

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

Scandura, J. M. Structural Learning I: Theory and Research. London/New York: Gordon and Breach Science Pub., Inc., 1973.

Scandura, J. M. Problem Solving: A Structural/Process Approach with Instructional Implications. New York: Academic Press, 1977.

Scandura, J. M. Theoretical foundations of instruction: a systems alternative to cognitive psychology. Journal of Structural Learning, 1980, 6, 347-394.

Scandura, J. M. Problem solving in schools and beyond: transitions from the naive to the neophyte to the master. Educational Psychologist, 1981, 16, 139-150.

Scandura, J. M. Structural (cognitive task) analysis: a method for analyzing content. Part I: background and empirical research. Journal of Structural Learning, 1982, 7, 101-114.

Scandura, J. M. Structural analysis. Part II: toward precision, objectivity and systematization. Journal of Structural Learning, 1984, 8, 1-28. (a)

Scandura, J. M. Structural analysis. Part III: validity and reliability. Journal of Structural Learning, 1984, 8, 173-193. (b)

Scandura, J. M. Structural learning: the science of cognitive, instructional and intelligent systems engineering. Journal of Structural Learning, 1985, 8, i-ii.

Scandura, J. M. PRODOC: The PROfessional self-DOCumenting programming productivity environment. Journal of Structural Learning, 1986, 9, 101-105.

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

Scandura, J. M. and Scandura, A. B. Structural Learning and Concrete Operations: An Approach to Piagetian Conservation. New York: Praeger Sci. Publ., Inc. 1980.