Cognitive Technology and the PRODOC re/NuSys Workbench™: A Technical Overview

JOSEPH M. SCANDURA
University of Pennsylvania

(Received May 1991; in final form June 1991)

INTRODUCTION

In this overview, we introduce the PRODOC™ family of design, development and maintenance systems and the cognitive approach to systems engineering on which it is based. Special attention is given to some of the ways in which PRODOC may be used to renew and recycle existing systems.

PRODOC™ is based on a cognitive approach (CogApp) to systems design, development and maintenance (Scandura, 1987, 1990). Stated succinctly, this cognitive technology 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. As we shall see below, 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 and macros (with new systems) and/or salvaged code (as a result of re-engineering).

The article is divided into five sections. Section 1 discusses the role of PRODOC in the contemporary Computer Aided Systems Engineering (CASE) world. In addition to basic methodologies, different notational schemes are discussed at some length with emphasis on FLOWforms. Section 2 deals with analysis and design. The major emphasis is on using PRODOC for requirements, specifications, high level design, checking designs and rapid prototyping (including a variety of subtopics ranging from screen prototypes to simulating high level designs and text coverage).

Section 3 discusses the use of PRODOC in implementing systems. First, we discuss the process of testing from the top down. Then, three modes of implementation are

---

1 Ed. Note: The PRODOC re/NuSys Workbench™ draws heavily on Dr. Scandura's basic research but was developed under the auspices of Scandura Intelligent Systems. For more information contact Scandura Intelligent Systems, 1249 Greentree, Narberth PA 19072.
discussed: developing low level routines in pseudocode, rapid program development (generating full source code from high level designs) and the use of high level system designs and reusable resources. Finally, we show how PRODOC automatically supports environment customization and conditional compilation. Section 4 deals with the maintenance and enhancement of existing systems. Reverse engineering, restructuring, renewing old designs, conversion to more modern languages and the generation of reports are all considered.

Section 5 summarizes the cognitive perspective on re-engineering, including a summary of design, implementation and re-engineering processes—all represented succinctly in FLOWforms. These processes are also integrated in an overall system renewal FLOWform.

1 PRODOC™ IN THE CASE WORLD

In describing their products, CASE vendors often confuse the terms “methodology” and “notation”. What are typically referred to as methodologies are really notational schemes. In fact, there are only two widely recognized development methodologies: variants of the so-called “waterfall” and (newer) “rapid prototyping” models.

Relative Advantages of the Waterfall and Rapid Prototyping Models

Figure 1 summarizes key stages in the waterfall approach to development. It is implicitly assumed that various phases of the life cycle are sequential in nature. It is widely recognized, however, that changes later in the life cycle typically require

The WATERFALL Model of the Software Development Life Cycle

- requirements
  (specify what system is to do)
- specification
  (specify how system is to do it)
- high level design
  (logical design of system)
- detailed design
  (language specific design of software)
- coding
- testing and integration
  (module, integration, system, user)
- maintenance
  (Fix bugs, add enhancements, etc.)

Figure 1 Stages of the waterfall model.
modifications to earlier work. For example, changes made at the detailed design and implementation stages frequently requires changes in data flow diagrams and entity-relationship diagrams developed during analysis with front end CASE tools. While it is theoretically possible to coordinate changes, this is rarely done in practice. Results at various stages of the life cycle invariably get out of sync over time.

Variations on the waterfall model fall into two general categories depending on whether data analysis is done first and then process analysis, or whether process analysis is done first and then data. In the waterfall model various phases of development are assumed to take place sequentially, with testing coming after implementation. Testing after implementation has an important limitation. The number of paths through a system becomes progressively larger as the system becomes more complex. In particular, the number of paths tends to increase exponentially with the number of independent decisions introduced. With even relatively small systems, the number of tests required to test all possible paths (even once) quickly becomes impractical, and soon impossible. This is the well known “combinatorial explosion” problem. On the other hand, if one could test underlying logic from the highest levels of abstraction, complexity only goes up additively with complexity. Implications of this difference are hard to overestimate. In an example cited by Scandura (1990), \(2^{100}\) tests would be required if all testing were done after implementation, and only 301 tests if done from the top-down. The order of magnitude of improvement is dramatic, turning an impossible task into something which is quite feasible.

Rapid prototyping comes in a variety of flavors. Common variations include: rapid methods for constructing user interfaces, often with branching between screens based on user inputs. Considerably less attention typically is given to executable specifications and simulating the underlying logic needed to implement actual systems.

System logic must be tested beginning from the highest levels of abstraction in order to get the full benefits described above. As traditionally practiced, rapid prototyping has some serious disadvantages in this context. There is a tendency in rapid prototyping, for example, to add progressively more patches to achieve desired user interfaces. Ultimately, this leads to a breakdown in underlying logic with the developmental process grinding to a halt.

The relative advantages and disadvantages of following the waterfall and rapid prototyping models are summarized in Figure 2. Notice that advantages of the waterfall model correspond one-to-one with limitations of rapid prototyping—and vice versa. Advantages of rapid prototyping correspond to limitations of the waterfall model. This suggests the desirability of combining the advantages of both models.

Variations on just such an approach have been proposed recently by several investigators (e.g., Boehm, 1988; Yeh, 1990; Scandura, 1990). Depending on emphasis and personal preferences, this new methodology has been variously referred to as the spiral model (Boehm, 1988), specifications programming (Yeh, 1990) and the cognitive (abstract) approach (Scandura, 1990). Each variation combines aspects of both data and process oriented design and rapid prototyping beginning at a high level of abstraction. The cognitive approach also makes explicit provision for re-engineering and code reuse.

The PRODOC re/NuSys Workbench™ is a full life cycle design, development and maintenance system which supports both data and process oriented methodologies
as well as rapid prototyping. It draws inspiration from Scandura’s (1987, 1990) cognitive approach to systems engineering and re-engineering. As a tool, the PRODOC re/NuSys Workbench™ is unique in integrating beneficial aspects of the above methodologies.

The following sections provide more conceptual background both as to how PRODOC™ relates to other contemporary thinking and notation schemes. More importantly, they also show how to actually use PRODOC™ in the analysis and design, implementation and maintenance of systems.

** Limitations of “Bubble Charts”**

‘Bubble chart’ representations have several important limitations.

a) ‘Bubble charts’ look entirely different from corresponding detailed designs and source code. The schism is inherent: a dichotomy necessarily exists at some level of analysis.

b) ‘Bubble charts’ are best used for simple high level overviews of abstract data and processes on that data. ‘Real time’ extensions can accommodate procedural sequence and control issues, but they add a level of complexity at odds with the original goals of visual clarity.

c) Front end CASE tools display successive refinements of ‘bubble charts’ in different rectangular windows. This not only limits the number of levels which may be seen at one time (inherent in the size of the display screen) but it becomes increasingly difficult to figure out which refinement window goes with which higher level element.
Even experienced CASE users find it difficult to find their way around in more than three or four levels of refinement at a time.

The diagrams in Figure 3 correspond to the three basic types of structured refinement (sequence, selection and iteration) and are included for comparative purposes.

**Other Alternatives**

If we want to represent all levels of process abstraction in a uniform way, then we are forced to look elsewhere. Action diagrams (e.g., Martin and McClure, 1988) and Warnier-Orr diagrams are perhaps more viable in this context. They effectively use embedded brackets in one form or another to mark off various levels of abstraction. In principle, one could mark successive brackets into a tree-like structure which could effectively contain higher as well as lower level information.

The main limitation of these representations, however, is that they say nothing visually about sequence or control. They are essentially just structure outliners. Information pertaining to sequence or control must be added to the verbal descriptions—thereby confounding abstraction with sequence and control issues.

Nassi-Schneiderman diagrams are better in this respect because sequence and control are dealt with visually. As Figure 4 illustrates. Nassi-Schneiderman diagrams are easily understood by any analyst, systems designer or programmer. Sequential control can be introduced more naturally than in data flow diagrams—see Comparing Notations.) Indeed, recent research by Professor Scanlon (1987) at the California State University shows that 75–80% of students learning how to design systems strongly prefer Nassi-Schneiderman diagrams over English-like pseudo-code.
Since Nassi-Shneiderman charts have been around as long as 'bubble charts,' why are they not more widely used? Perhaps the major consideration in this regard is that they are hard to draw and even harder to change. It is much easier to connect bubbles (with lines) in different ways than to make corresponding changes in a Nassi-Shneiderman chart. Today, with the advent of CASE this should no longer be a problem. In principle, Nassi-Shneiderman diagrams might be redrawn automatically just as well as 'bubble charts.'

Another important factor that has worked against Nassi-Shneiderman diagrams is that selections are handled by splitting rectangles vertically (see Figure 4B). This limits severely the number of levels of refinement that can be displayed without having to open a new window. A third major limitation is that Nassi-Shneiderman charts do not deal with data.

FLOWform Notation

As shown in Figure 5 (also see Scandura, 1987). FLOWforms combine advantages of bubble charts and Nassi-Shneiderman diagrams without their disadvantages. Perhaps most important, FLOWforms support contextual windowing. The overall context is always clear no matter how many levels of refinement are displayed. FLOWforms have the further advantage of making it possible to see higher-and-lower level abstractions at the same time and in the same context.

Experience with the PRODOC re/NuSys Workbench™ demonstrates that FLOWforms can be created and changed more quickly than bubble charts. Informal experiments further suggest they are easier to understand than either bracketing systems or bubble charts. Moreover, modifications can be made at any level with direct visual traceability to all effected levels of abstraction. That is, one can describe systems in English and in context at whatever level of abstraction is desired (see Figure 5).

FLOWforms constitute basic building blocks. The systems constructed therefrom can be of arbitrary size. Higher level FLOWform modules contain links to (other) FLOWform modules they use, much as higher level routines are related to called subroutines. Changes in one FLOWform do not affect other FLOWforms, unless one explicitly changes links in the 'calling' FLOWform. This has important benefits.

![FLOWform Notation Diagram](image-url)

Figure 5  A. Sequence. B. Selection. C. Iteration. D. Sequence with higher level annotation.
for such things as configuration management and version control which are addressed elsewhere.

As shown in the next section, FLOWforms can also be used to represent overall relationships between FLOWform modules. Calling hierarchies or structure charts are perhaps most widely used in this context.

Comparing Notations

Figure 6 shows a typical data flow diagram (taken from Yourdon, 1989) and an equivalent process or procedure FLOWform. Although Yourdon refers to the clarity of the data flow notation, notice that it is not necessarily clear where to begin reading the data flow diagram. On the other hand, one always reads FLOWforms starting from the top. Notice also that the procedure FLOWform makes explicit provision for control structures. (The inclusion of control structures in traditional data flow diagrams requires notational extension as shown in Figure 7.)

Figure 7 (from Yourdon, 1989) shows traditional and FLOWform representations of a simple "real time" system. The "bubble chart" shows a traditional data flow diagram with real-time extensions. Equivalent FLOWforms for process and data appear below. Both the operations (solid circles) and control structure (broken circle) along with data flows in the traditional data flow diagram are represented in the procedure FLOWform at the top (with data in parentheses). The data FLOWform just below depicts the inputs and outputs explicitly.

There are similar parallels with regard to data. In Figure 8 we see a simple entity-relationship diagram on the top and the equivalent data FLOWform on the bottom.

Notice that all elements in data FLOWforms are represented in essentially the same way. "Entities" correspond to terminal elements. At the next level are negotiation relations which involve these entities. At the top level is an "agent" relation which handles the first order relations. This process can be extended as far as one wants without change.

In Figure 9, we illustrate a simple context diagram indicating relationships between the system and the outside world. Traditional notation is used at the top with equivalent process and data FLOWform representations below. The system itself is represented by the AJAX CORPORATION box in the data FLOWform (rather than as a circle in the context diagram). Input and output entities are listed under "DOMAIN" and "RANGE" respectively. "Customer" (a complex entity) and "IRS" serve as both input and output and are represented in the FLOWform as clones.

In Figure 9B, the AJAX CORPORATION system is depicted in the top (procedural) element. Under DOMAIN are elements in the outside world which provide input to the system. Elements under RANGE receive output from the system. Listed below the dots (.) are "clones"—elements contained in both DOMAIN and RANGE.

The procedure FLOWform below (Figure 9C) shows an initial expansion of the top level AJAX process.
Process book orders

<table>
<thead>
<tr>
<th>WHILE valid_order</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO Recieve customer order.</td>
</tr>
<tr>
<td>CASE parallel_action OF</td>
</tr>
<tr>
<td>shipping</td>
</tr>
<tr>
<td>Process order details.</td>
</tr>
<tr>
<td>Get books from warehouse and ship to customer.</td>
</tr>
<tr>
<td>billing</td>
</tr>
<tr>
<td>Process billing information (i.e., name, address, etc.)</td>
</tr>
<tr>
<td>REPEAT Send invoices, statements.</td>
</tr>
<tr>
<td>UNTIL payments_collected</td>
</tr>
</tbody>
</table>

Figure 6 A. Typical Data Flow Diagram (Yourdon, 1989) and B. Equivalent FLOWform.

Relationships between modules: All of the above FLOWforms use a common box-like notational scheme for both data and process. This FLOWform notation is uniform throughout and is equally applicable at any level of abstraction from the highest levels of specification to the lowest levels, including actual code.

None of the above notations, however, represent relationships between modules. Calling hierarchies are frequently represented as tree-like diagrams (see Figure 10A).
Figure 7  A. Simple Data Flow Diagram with “real time” extensions (Yourdon, 1989). B. Equivalent process FLOWform. C. Data FLOWform.
In this calling hierarchy, subroutine A is assumed to call subroutines B and C. Subroutine B in turn calls subroutine D and so on.

Additional notation arrows and/or labels are frequently added to designate parameters being passed. Depending on specifics, these notations are more or less easily interpreted.

The FLOWform equivalent of the above calling hierarchy is shown in Figure 10B.
Figure 9A  Context diagram using traditional notation.

Figure 9B  Context diagram using FLOWforms.
Calling hierarchies provide important information concerning module relationships. The inclusion of parameter passing information helps even more. But, why stop there? Understanding system relationships often requires knowing what global variables are used by various modules, or perhaps how the modules are categorized. It is easy to include such information in FLOWforms by simply including additional levels of refinement. The FLOWform in Fig. 11 illustrates several kinds of relationships between Ada modules. The MONITOR-TEMPERATURES procedure body

Figure 9C High level expansion of the AJAX CORPORATION.

Figure 10A Sample calling hierarchy.
coordinates interactions between four tasks, each of which has one or more entry declarations (analogous to parameters) and hierarchies of tasks it uses.

As we shall see below, arbitrary system FLOWforms representing overall relationships can be constructed automatically from individual modules under high level program control.

2 ANALYSIS AND DESIGN

FLOWforms represent one notational scheme for representing systems at various levels of abstraction. What one actually puts inside elements of this notational scheme (or any other) is quite arbitrary. At one extreme, we frequently use ordinary English (or French, German, etc.). At the other extreme, we may use an explicit formal language with a precise syntactic construction.

Requirements

Requirements concern what a system is to do—what data it is to produce and under what circumstances. This data can be described generally at a very high level of abstraction or in very precise terms with an explicit syntax and semantics. Although there are exceptions, requirements typically are expressed in ordinary English. In PRODOC™ this may be done by entering English descriptions, optionally along with the name of the data elements and/or relationships involved in a data FLOWform as shown in Figure 12.

Alternatively, requirements may simply be entered into a procedural FLOWform as text.

Specifications

Specifications deal essentially with how requirements are to be met—what the system is to do, and when (i.e., the conditions under which things are to be done). Again, specifications are typically, although not necessarily, stated in English. As with data
Figure 11 Ada system FLOWform showing relationships between modules.
FLOWforms, process or procedural FLOWforms represent information at various levels of abstraction. One typically begins by describing the highest level elements or processes and progressively refines them so as to expose increasing amounts of detail.

Figure 15 shows several levels of refinement in a procedure FLOWform. The highest levels are in English; lower levels use the syntax "operation (data_structure 1, ... )" and, hence, constitute elements in a high level design.

High Level Design

The purpose of a high level design is to expose the underlying system logic. Consequently, a somewhat more formal language is desirable. On the other hand, the form of representation chosen should be as simple and easy to read as possible, consistent with providing an adequate means of representation.
In PRODOC™, data is entered into various fields of a data element:
The name of the data element (or relationship) appears first within square brackets.
Next comes the data type within curly brackets "{" and "}". The third field
is an expression representing the functional relationship between the given element
and other data elements from which its value is derived. The next field appears after
a colon within square brackets [ ]. This expression specifies how the data element
may inherit its value from elements higher in the data hierarchy. Finally, the
remainder of the data element contains the value assigned to that element. All except
the "::" are optional. The figure below makes this explicit.

```
[ name ]( type )eval_fn(A,B); [ inherit_fn(C,...)]value
```

Figure 13  Fields in a data element.

Unlike data elements, process elements have names only when they refer to separate
FLOWforms (e.g., the name field may contain the name of a referenced TEXT or
SUBTREE file). The types field is used in this case for the directory, path and filename.
(Corresponding FLOWforms are available at the press of a key.) The evaluation
function field is used to specify the kind of procedure structure (e.g., SEQUENCE,
IF, UNTIL, etc.). This field also is used to indicate that the procedure element names
or corresponds to another FLOWform (SUBTREE) or TEXT file. (In this case, the
name of the FLOWform or text file is given in the types field.) The square brackets
after the colon contain selection alternatives in CASE structures.

Operations along with the data being operated on (i.e., statements) go in the value
field after the colon. The syntax used is simply "operation (parameter_1, ..., parameter_n)"
where each parameter may, in turn, be defined as an operation on
data. As a convenience, "::=" also is allowed for assignment. The example below
shows a typical statement.

```
X ::= operation1(A, operation2(B,C))
```

Figure 14  Sample statement in a procedure FLOWform.

PRODOC™ supports either data first analysis, where the data is defined first and
then the processes acting on the data, or process first analysis, where the process is
defined first and then the data. Figure 15 shows three levels of procedural refinement.

When data is developed first, PRODOC™ checking facilities (see below) can be
used to determine consistency between procedural and data FLOWforms. In particu-
lar, if a data element is referred to in the procedure but not found in the data
FLOWform, the user will be required to declare it. Conversely, if there are data
elements which are not used in the procedure, the user will be given an opportunity
to delete them as redundant. When a procedure is first developed, the same syntax
checking process identifies each of the identifiers in turn (both process and data
identifiers) and requires the user to specify what they are (e.g., an included file,
variable, etc.). PRODOC™ then automatically creates a corresponding element in
the data FLOWform.
Data and procedural FLOWforms also may be developed in parallel beginning at the highest level of abstraction. As noted above, there are many advantages of this approach. Scandura (1990), Boehm (1988) and Yeh (1990), among others, strongly recommend a parallel approach to data and process even though many of the traditional software engineering "methodologies" assume either data first or process first analysis.

**Checking designs**

The PRODOC re/NuSys Workbench™ checks designs automatically. Several levels of checking are supported. First, it is impossible to construct an invalid process or data structure. For example, it is impossible to define an isolated data element; PRODOC automatically insures a valid connection. Similarly, it is impossible to construct a loop without a body or an "if" construct without a corresponding "Then" alternative.

Second, PRODOC's syntax checking not only checks for correct syntax but for consistency between procedural and data FLOWforms. For example, if a data element is referred to in the procedure but not found in the data FLOWform, the user will be required to declare it. PRODOC™ then automatically creates a corresponding element in the data FLOWform.
Conversely, if there is a data element that is not used in the corresponding procedure, the user may choose to keep, delete or otherwise rectify the situation. All of this can be done interactively.

**Rapid Prototyping**

The term "rapid prototyping" normally refers to the rapid construction of user screens, and often includes interactive machinery for switching between screens. Rapid prototyping, more generally, refers also to the simulation of underlying logic and to "rapid" methods of implementation. Executable specifications and the automatic generation of code from high level designs also play a major role.

**Screen Prototypes:** To prototype user screens it is sufficient to create a procedure FLOWform, refining the top level into a simple sequence of steps. Elements of the sequence contain references to the individual prototype screens. Specifically, each box or terminal element in the FLOWform will contain a statement of the form "display (screen_name)".

The PRODOC Simulator environment is used to display system screens. When the simulator gets to the statement, "display (screen_name)" it simply puts up whatever value has been assigned to screen_name. The screen may be formatted in any way using optional attributes (e.g., for color) and/or carriage returns, tabs, etc.

**Branching Among User Screens:** In order to effect branching among screens, we get input from the user and add necessary logic to the procedure FLOWform. Hence, we might insert a “get_input” node immediately after each display statement. When the simulator gets to this operation it will wait for the user input. When return is pressed it will store whatever the user has input as the value of the input parameter element. Subsequent branching then may be affected by imbedding one or more display statements, for example, in an if/then statement (or loop).

**English Specifications:** A lot more goes on in a program, of course, than simply branching among user screens. Specifying how a system is to operate involves describing things the system is to do and under what conditions.

Having specified a system, PRODOC will simply trace through the various paths in turn according to answers input by the user at each decision point (e.g., whenever an “If” or “loop” condition comes up).

**Simulating High Level (Logical) Designs:** More exacting simulation is possible when statements in a procedure FLOWform are put in PRODOC’s library or prototyping syntax and corresponding data elements have been added (e.g., by syntax checking or construction of a data hierarchy) in the corresponding data FLOWforms. In this case, whenever the PRODOC Simulator comes to an operation that it does not understand it displays the current values of the various parameters and asks the user to simulate performance of the operation and update the values. These values are retained. There, they play the same role in simulation as they would during
execution of an actual system. There is no difference whether procedures are viewed as transactions on a database or as a real-time or embedded system.

Processing Parallel Procedures: It is possible to simulate the execution of parallel procedural designs in PRODOC by constructing a case statement (where each of the case alternatives corresponds to one of the parallel procedures) and embedding the case structure in a repeat/until loop (with an appropriate repeat condition).

Object Oriented Simulation: The PRODOC Simulator also can be used to simulate (and/or execute) objects which combine data and process.

Using the "Display_Structure" Library Rule: Whereas the "display" rule is used to display the value of single data elements, the "display_structure" rule can display the collective contents of an entire data structure (or any substructure thereof). This makes it possible with a single statement to display an entire structure on the screen, various parts of which may be rearranged subsequently as desired. As in most object oriented formulations, various data elements inherit display characteristics from data elements higher in the data structure hierarchy.

Simulating Data Execution: Asynchronous processes may be simulated via data execution. In particular, evaluation function fields and inherent function fields of data nodes may be used to define functional dependencies among various data elements.

Suppose for example, that a data element is defined in terms of another data element. In this case, it is not necessary to have a prior statement in the procedure defining the value of that data element. This information is already available in the data. In this case the user would be asked to indicate the value of the data given the value of the inputs to the defining function. That result then would be input to the procedure operation which would be simulated as usual.

Simulating Interrupts: The same machinery may be used to simulate interrupts. Consider an operation of the form "Operation (effective_parameter)". In this case, "effective_parameter" in the data FLOWform would be defined as a function of both the actual data and the interrupt--in the form "functional_operator (data, interrupt)" where data and interrupt are data elements in their own right. If the interrupt is active, an exception value would be returned via the effective parameter whereas if it were inactive, the normal data value would be transmitted.

State-Transition Diagrams: PRODOC™ does not represent state-transition diagrams statically in terms of tables or directed graphs. It does, however, make it possible to dynamically simulate structured state-transition diagrams. In particular, "debug" mode in the Simulator displays all (or any part) of the state of the system before and after each operation.

Test Coverage: This option provides a visual record of all steps in a design executed during simulation. A record also is kept of the number of times various steps in a design have been executed.
**Test Data:** Updated data structures can be saved for subsequent testing at the conclusion of each simulation session. In turn, test data structures can be used as (file) inputs during subsequent simulations sessions. Test data structures can also be constructed by editing data FLOWforms directly and saving the results to disk. In either case, only operational changes to the data need be entered during simulation.

**Concluding Comments:** The above examples are not meant to be exhaustive. The PRODOC Simulator can be used for a wide variety of possible simulations depending only on the imagination of the user. Hardware designers at IBM, for example, have found PRODOC's Simulator valuable in designing and testing high level behaviors of complex hardware designs prior to committing the designs to silicon.

### 3 USING PRODOC™ TO IMPLEMENT SYSTEMS

Assume we already have a system specification and/or high level design. These specifications and/or high level designs may be represented in: a) a separate document; b) data flow diagram and/or entity relationship notation, or c) FLOWforms as described above. In the first case (a), one may choose to begin by constructing a FLOWform reflecting specifications and/or designs described in a separate document.

Essentially the same procedure would be used in case (b) where a different CASE tool has been used for front end design. Alternatively, a simple bridge might be built (or purchased) from that tool to PRODOC. PRODOC's import/export facilities and extensible library provide facilities for building such bridges. Prebuilt bridges are available from other companies (e.g., ASTEC).¹

FLOWforms correspond one-to-one with the terminal elements in a typical bubble-chart (i.e., data flow diagram). That is, high-level descriptions in the terminal bubble charts correspond to the highest level descriptions in FLOWforms. Corresponding “minispecs” are equivalent to FLOWform contents. The higher levels in data flow diagrams also may be configured as FLOWforms where one chooses to work uniformly in the FLOWform representation.

Starting with FLOWforms to begin with as in case (c) has the obvious advantage that one can proceed from specification and high level design to detail design and implementation without any change in the underlying representation.

**Testing from the top down**

In the previous section, we discussed advantages of testing high level specifications and designs long before actual implementation. Figure 16 illustrates the process at

---

¹ FLOWforms correspond one-to-one with the terminal elements in a typical bubble chart. That is, high level descriptions in the terminal bubble charts correspond to the highest level descriptions in FLOWforms. Consequently, there is relatively little to import. Short descriptions in terminal bubble charts can frequently be entered directly as rapidly as importing the information automatically. The only case where it makes any sense to import information into individual FLOWforms is where terminal bubbles have already been described as mini-specs. In this case, the mini-specs may be uploaded into FLOWforms by using the generic source code capabilities.
one level of abstraction (only process structures are shown). Elements in the FLOWorm at the top use PRODOC's neutral syntax. This syntax is both simple and familiar. It is common to most modern programming languages, and may be viewed as that which is common among Ada, Pascal, and C.

One step in simulating the process logic is shown immediately below the FLOW-form at the top. (Not shown in the figure are the data structures.)

Thoroughly testing a system design (both data and process) at any one level of abstraction provides a sound basis for further refinement. Since overall logic has already been tested, subsequent testing can reasonably be limited to the individual refinements. This approach dramatically reduces the number of required tests.
Implementation

The first step in implementing specifications or high level designs is to make contact with reusable routines and/or to satisfy oneself that individual components can easily and reliably be implemented. FLOWform specifications and designs may be implemented in three basic ways depending on implementation goals and constraints: One, FLOWform specifications and/or designs refer to low level routines which are to be implemented with maximum flexibility (i.e., using the full power of the target programming language). Two, implementation of high level specifications and/or designs is to be accomplished as quickly as possible, where code efficiency and flexibility are secondary concerns. Three, high level system specifications and designs are to be implemented with respect to existing libraries of reusable routines in some target language.

Developing Low Level Routines

In the first case, the programmer knows exactly what he wants to do and how to do it and has done similar things before with little difficulty. In these cases, the designer/programmer typically assigns the target language (C, Pascal, Ada, Fortran or COBOL) to the FLOWform from the very beginning.

Elements in the FLOWform design are successively refined until pseudocode in the target language is entered in the terminal elements. A programmer who knows one of the re/NuSys languages already knows the corresponding pseudocode. C pseudocode, for example, is simply C without the braces and semicolons used by the compiler for parsing. This irrelevant information is replaced by the visual FLOWform.

Successive Refinement of C function

![Figure 17 Successive refinement of C function.](image)
representation making it easier to see the essential concepts without any loss of flexibility.

When satisfied with a pseudocode implementation, the implementer chooses the Code Generator environment and presses "G" for generate. Declarations can be entered manually under variables, types, etc. with the number of FLOWform sections and their names depending upon the languages assigned to the FLOWform.

![Modular System in C](image)

**Figure 18** Modular system in C.
Alternatively, syntax checking capabilities in the ScanFLOW Designer can be used. PRODOC will, in turn, identify each of the identifiers (operations and data) and ask you to select their appropriate categories. In the process, PRODOC automatically adds all the information needed to generate correct declarations to the data FLOWform.

**High Level Systems (Rapid Program Development)**

We begin with a high level FLOWform design, with "Library" as the chosen language. In this case, no transition from high level design to low level design is required. Assuming you have entered statements in the library procedure FLOWform in the usual PRODOC syntax, simply continue the same process of refinement until you make contact with the rules available in PRODOC's library.

By now it is obvious that operations and their parameters in a FLOWform may be described at any level of abstraction. It is equally true, although perhaps not so obvious, that a full range of data abstraction is also available. There are times, for example, when one might want to insert a loop condition to the effect "perform the body of the loop until 'done'." The question is how to determine whether "done" is true or false. In normal programming this might require some complex computation defined on other variables. This supportive but conceptually superfluous code would normally be placed in the procedure.

In the PRODOC Simulator, however, the value of "done" may be determined by executing (or simulating) an evaluation function where the value of "done" is defined in terms of other data elements. In this case, what we do is move some of the procedural complexity over to the data where it more naturally belongs.

In most cases one would not want to actually use the PRODOC Simulator to execute a system. PRODOC's High Level Design C Generator takes high level designs implemented in terms of PRODOC library rules and generates C code and data structures therefrom ready for the compiler. For this purpose, the High Level Design Generator assumes all code is procedural—that is, that data execution functions have been moved to corresponding positions in the procedural code.

---

**Figure 19** Hierarchical structure of the system.

This figure represents the hierarchical structure of the system. Notice that by definition of functions in the C language, main (or any other function) can call `sqr` directly (dashed line).
The High Level Design C Generator supports modular development, that is, high level FLOWforms can call or use other FLOWform modules. It also is possible to add your own routines.

Library Rules, the Library Generator and Higher Order FLOWforms

Scandura Intelligent Systems has also developed a Library Generator which greatly simplifies the task of incorporating new "library" rules into PRODOC. PRODOC's standard library contains a wide variety of rules. Most notable (in addition to standard input/output, arithmetic and assignment rules) are a variety of powerful higher order rules which are especially well suited to manipulating and/or constructing FLOWforms. FLOWforms constructed from such rules are "higher order" in the sense that they operate on and/or create (other) FLOWforms. FLOWforms themselves represent programs. In effect higher order FLOWforms can be used to create or modify program semantics under program control. This truly unique and powerful feature derives from our earlier basic research in structural learning (e.g., Scandura, 1973, 1974, 1977). Its implications for software engineering are only beginning to be felt.

High Level Systems and Reusable Resources

A third mode of implementation is to translate a logical design FLOWform into a FLOWform in the target language. Suppose you had a high level FLOWform design in PRODOC syntax. You also have a set of reusable library routines or macros (e.g., containing CICS calls) in one or more of PRODOC's pseudocode languages (currently Pascal, C, Ada, COBOL and Fortran). Assume that the high level design has been tested via simulation and that you are satisfied with the results. Also assume that each design component (terminal element of the FLOWform design) is a fully functional "black box." And, assume that these black boxes correspond to routines (or macros) available in some reusable library.

In this case, the chosen course of action would be to go to the PRODOC Translator and convert the high level design into the target language. What the translation does in this case is to take the library format in the procedural FLOWform and convert it to a new procedure FLOWform assigned the target language. Procedure structures are converted along with the individual statements as necessary. The overall process is represented schematically with respect to C in Figure 16.

Special attention during syntax checking should be given to the referenced routines and/or macros. The 'C' FLOWform at the bottom assumes availability of reusable 'C' routines corresponding to elements in the logical design (FLOWform) at the top. In this case, the 'C' FLOWform differs at the pseudocode level only in the loop condition. Local subroutines in a hierarchical language such as Pascal, C or Ada are

---

2 For most target languages, the translations are largely one-for-one with one library statement corresponding to a similar statement in the target language in the appropriate syntax. In the case of COBOL, there typically are several COBOL statements for each statement in the Library language.
normally assigned to the INCLUDED FILE section. These assignments automatically create links to the called FLOWforms. These links are used for a variety of purposes such as code generation, hardcopy, generating calling hierarchies, etc. (when the “included File” option is “on”).

Other linking mechanisms, namely “SUBTREE” and “TEXT”, are more general and especially useful in reusing macros in languages like COBOL and Fortran. These mechanisms make it possible to (recursively) add “subtree” FLOWforms or text files at any point in a data or process FLOWform—just as if the subtree or text were expanded directly in the context in which it appears.

**Environment Customization**

Subtree and text links are automatically used in code generation. This makes it easy, for example, to reference commonly used macros (e.g., CICS calls) which need not be independent subroutines. In effect, one can program at as high a level of abstraction as might be desired, the level depending only on the routine libraries or macros available for reuse. Rather than being restricted to a given set of available resources, as in most 4GL’s, PRODOC™ makes it possible to create one’s own high level language.

Individual subtree, as well as included file, FLOWform files also can be customized for particular hardware configurations. This reduces the need to maintain separate code for different hardware or operating environments. Customization is accomplished by labeling various elements in a FLOWform (e.g., >AS400>) and using the code generation option with environment customization (for a designated environment). Assuming all elements in the high level design have counterparts in the library, code generation results in a fully functional system that can be maintained entirely within the high level design.

Conditional compilation meta commands also can be generated automatically from labels (e.g., >DOS_only>) inserted into FLOWform elements. This is particularly useful in maintaining different versions of a software product written in a modern language (e.g., C. Pascal).

**4 USING PRODOC™ TO MAINTAIN OR ENHANCE EXISTING SYSTEMS**

Code maintenance takes from 70% to 95% of the typical data processing budget. Heavy maintenance costs derive from three quite different sources: One, to-be-maintained code generally includes a good deal of irrelevant information, most notably key words and compiler tokens like “BEGIN”, “END”, “;”, “)”, etc., used by the compiler for parsing purposes. From a cognitive perspective, this information

---

3 Library language structures are hierarchical: target data structures, with the exception of COBOL, are essentially flat. Data (e.g., variable) hierarchies in the library language correspond essentially to records of records ... of records. Hierarchical structures in a language like C. Pascal or Ada are realized by defining corresponding data types.
is irrelevant and often gets in the way of human understanding. Another source of difficulty derives from poor or inadequate formatting making it more difficult to encode (i.e., understand) overall structure. FLOWforms put pseudocode in an easy to understand visual environment while eliminating irrelevant information almost entirely.

Second, due to time constraints, code modifications and enhancements typically take the form of "patches" which tend to complicate code. Only rarely is code reorganized to achieve new or improved functionality, even when this would clearly be the best way to go. Over time, interactions between various portions of the code make it increasingly more difficult to introduce even the simplest changes. Complications of this type are usually far more difficult to handle than the first. Solution typically requires major system overhaul, if not completely new development.

A third type of problem derives from the linear structure of older languages like Fortran IV and COBOL 74. In this case, GOTO's are used not just for legitimate purposes (such as exiting from a loop) but because earlier versions of the languages (e.g., FORTRAN IV, COBOL 74) do not support arbitrary embedding as is the case in more modern languages and more recent versions (e.g., FORTRAN 77, COBOL 85). Trailing code segments (e.g., COBOL paragraphs and continue labels) are typically used in these cases. To understand such code, one is forced to study and interrelate two or more sections of code simultaneously. PRODOC support for conversion from older to newer languages is highly relevant in this context.

Three typical situations are considered below.

Reverse Engineering

Assume we have a body of reasonably well organized source code that we wish to maintain in PRODOC's hierarchical FLOWform environment. The advantages of doing so derive from simpler looking pseudocode and from the ability to view code specifications at whatever level of abstraction might be desired. Detail is inspected only where necessary and in the desired high level context. Moreover, pseudocode in FLOWforms is more easily modified and maintained. The ease with which code can be restructured, for example, reduces the temptation to add non-productive patches.

To reverse engineer source code, the first step is to inspect the code making sure it does not include any obvious non-standard additions such as meta-commands before the program heading, etc. Having done this, and having made whatever other adjustments may be required, PRODOC will reverse engineer the code, uploading it into FLOWforms as pseudocode. (Figure 20 shows a Fortran example.)

In the case of older languages such as Fortran IV and COBOL 74, it is desirable to put trailing code segments and shorter paragraphs in the proper hierarchical context. While optional, this is easily accomplished with the ScanFLOW Designer's restructuring capabilities. Experience shows that the results are well worth the effort. Restructuring (except to remove GOTO's) tends to be less critical in modern languages.

Having restructured the code, and/or in parallel with the restructuring, it is
Fortran Reverse-Engineering and Re-Structuring

C SUBROUTINE MYSUB
....
IF(D.LT.A) GO TO 25
B = Y
RETURN
CONTINUE
IF(D.LT.A-B) GO TO 45
IF(B.GT.Z) GO TO 35
B = Y
RETURN
35 B = W
RETURN
45 B = Z
RETURN
END

This is a Fortran subroutine that assigns different values to the variable B under different conditions.

The code above is automatically reverse engineered into a FLOWform. It is still unstructured, but now the visual FLOWform environment makes it a lot more understandable. Notice also the 'place holders' for the missing documentation (empty boxes with dotted lines).

UNSTRUCTURED FLOWFORM.

The code above is automatically reverse engineered into a FLOWform. It is still unstructured, but now the visual FLOWform environment makes it a lot more understandable. Notice also the 'place holders' for the missing documentation (empty boxes with dotted lines).

Figure 20  Fortran reverse-engineering and restructuring.

important to add appropriate annotation at various levels in the FLOWform hierarchy. In general, each high level node should describe the structure beneath it—that is, indicate its functionality, and perhaps what it operates on and what it produces. A common mistake in adding high level annotation is to anticipate details at a lower level in the description. It is far more useful to say only enough at each level to describe the overall functionality. The description should not make reference to individual components below the level at which one is working. The example at the top shows a poor description, the one below is much better.
“Select the largest element and put it first. Then repeat the process iteratively.”
“Sort elements in array A.”

Once the reverse engineered code has been properly annotated, experience shows that maintainers can much more easily understand the code as well as modify it. All modifications are done at the pseudocode (and/or design) level within the FLOWform itself. Source code is regenerated automatically as needed.

It is important to recognize that FLOWforms are like hierarchically arranged modules which collectively capture the essence of an entire system. One FLOWform can include links to other FLOWforms just as one subroutine can call other subroutines. Conversely, source code is automatically broken down into FLOWform modules during reverse engineering (i.e., the uploading process). Links between the FLOWforms are automatically inserted.

**Design Recapture**

The reverse engineering process automatically creates a system file consisting of included file references to all of the FLOWform modules formed during uploading. These references effectively link the system file to modules in the system. This process does not, however, reflect calling relationships among the modules.

A wide variety of relationships can nonetheless be recaptured in a two step process. First, each FLOWform must be checked to create links to the FLOWforms used. This process also makes it possible to distinguish between various kinds of declarations (e.g., variables, types, etc.) and, more importantly, between those declarations which are global and those which are local. Second, this information is readily available via the Simulator’s high order rules. These rules make it easy to detect FLOWform characteristics, to modify them under program control and/or to automatically create entirely new FLOWforms. Calling hierarchies are one type of system FLOWform which can be created in this way. But, traditional calling hierarchies represent only partial information. Why not include information about the parameter types, or global variables which are used? Such information is readily available.

**STRUCTURED FORTRAN SUBROUTINE**

After reverse engineering, the next step is editing the resulting FLOWform using the powerful system editing features within the ScanFLOW Designer. The structured FLOWform resulting from this process, as applied to the FLOWform in Figure 20, is shown in Figure 21. Notice how we removed the ‘GOTOs’ and replaced them with nested structures. (Note: In the case of Fortran, the optional PSTRUCT utility can be used to automatically restructure the Fortran source code before reverse engineering.)

**Renewing Old Designs**

All too often the code we seek to salvage is poorly structured, so called “spaghetti” code. We can reverse engineer such code into FLOWforms as before and doing so
Figure 21  (A) Structured Fortran subroutine and (B) The same FLOWform after translation to Ada.
will have the same visual advantages. It will not, however, eliminate distant cross references between various sections of code. Indeed, in the case of COBOL there may even be references within one FLOWform to other elements in entirely different FLOWforms. Largely for this reason, COBOL FLOWforms are allowed to include more than one subroutine (i.e., paragraph).

What is needed in this case is a new clean design which captures both the essence of the current design preferably together with desired extensions on an existing "wish-list". The PRODOC ScanFLOW Designer and Simulator combination provides strong support for both development and testing new designs independently of the underlying code.

Creating a new design, of course, even one that has been thoroughly tested, is only part of the story. Inspection usually reveals that a high proportion of code, in even the worst organized source code, is reasonably well organized and quite salvageable. Typically, this includes most lower level routines. Often rather large blocks of code in the original system are in reasonably good shape. This code can be put in even easier to maintain form by reverse engineering as described above. Because the new design obviously relates to the current system, the elements in the new design invariably map fairly cleanly into the reverse engineered components. Indeed, both the new design and reverse engineering should be done in parallel with mutual cognizance of the parallel activity. The basic process is represented in the schematic below (see Figure 22).

Needed components such as "New Module" would be constructed as described above.

Once the high level design has been tested in the library language, it must be translated using the PRODOC Translator into the target language. This is followed by "fine tuning", declarations generation and code generation. Assuming a consistent naming scheme for elements in the design, syntax checking will result in the automatic creation of included file links with the reusable code.

Conversion to More Modern Languages

Conversion to more modern languages occurs in two general contexts. The first involves conversion from an older to a more modern version of the same language. For example, from COBOL 74 to COBOL 85 or Fortran IV to Fortran 77. The solution in this case is to use the ScanFLOW Designer or PSTRUCT to restructure the code. (In the former case, the older form of the language must first be reverse engineered into FLOWforms.)

The second type of conversion arises when one chooses to move from an older to a more modern language—for example, from COBOL to C or from Pascal or Fortran to Ada. In this case, the solution is to first reverse engineer the older code into a hierarchical FLOWform. If the conversion is from an older form of COBOL or Fortran, it is normally a good idea to restructure the code before translation. The re/NuSys Fortran restructurer (PSTRUCT) can be used for this purpose.

At this point one can go to the PRODOC Translator and choose the desired target language. Annotating higher levels in the FLOWform also is typically required. This
PRODOC re/NuSys Workbench

Software Recycling: Renewal via redesign, re-engineering and code reuse

1. "Spaghetti" Code: Most software systems come to look like this as a result of on-going maintenance. Notice that most of the individual modules are fine, and can be reused. The whole system, however, is rather disorganized. (Most new design ideas also start like this.)

   IF A > B AND C > D THEN
      Replace A with B!
      Temp = A
      A = B
      B = C

2A. Re-engineer Code: Reverse engineer code from the existing system into modular FLOWforms. You easily understand module structure and conveniently make needed repairs. (This step is unnecessary in developing new systems.)

2B. Redesign System: Redesign the desired system at a high level, using PRODOC™’s universal 4GL. Test your design for logical errors, simulating process and data. At this point you should have a hierarchically structured system. (Step 2B is optional when the original structure is acceptable.)

3. Reuse Modules: Map reusable modules from your old system into the new high level system design. Finally, use PRODOC™ to design and develop missing low level routines. Experience shows that from 50% to as much as 95% of existing modules are reusable.

Figure 22 Code Recycling and Renewal: Combining New Development with Reverse Engineering.

can be done either before the translation or after since the basic hierarchical design will be the same in both cases. No translation, of course, is 100% correct (although the Library to target language translations come close to that ideal). Hence, some modification of the translated code will typically be required. The result of translating the Fortran example of Figure 21A into Ada is shown in Figure 21B. Notice that the design documentation at all abstraction levels is carried along automatically.
Semantic Modification of Translated FLOWforms Under Higher Order Program Control

Experience shows that the number of different kinds of translation problems in any particular case is relatively small. Moreover, the re/NuSys Simulator includes powerful higher order rules which are ideally suited for making such modifications. Consequently, it will often be possible to correct many (if not all) of the most common translation errors automatically under program control.

The PRODOC Repository and Report Generation

PRODOC\(^\text{TM}\) 's repository is distributed, consisting of any number of linked FLOWforms. FLOWforms may be linked by reference in the INCLUDED FILE section of a calling FLOWform or by direct reference. In the latter case, FLOWforms may be referenced from a given FLOWform by designating an element of the given FLOWform to be a SUBTREE element. Text files may be similarly referenced by designating a TEXT element. Both included files and subtree files may be called recursively.

PRODOC's report generation capabilities fall into four general categories:

*Printing Hardcopy: PRODOC\(^\text{TM}\) 's "Hardcopy" option can be used to create printed versions of FLOWforms, either singly or collectively, with essentially any printer and in essentially any configuration.*

*Exporting and Importing ASCII Files: Most CASE tools provide an export/import capability via ASCII files. Most front-end tools repositories, as well as the ASCII files which reflect them, contain positional information as well as essential entities and relationships. Positional information is typically needed to reconstruct the corresponding visual representations. This makes it difficult to exchange data with other tools without losing critical information.*

PRODOC\(^\text{TM}\) avoids this problem entirely because it does not use or require any positional information. Only essential information concerning entities and relationships is required to automatically and dynamically construct FLOWforms. Simply structured ASCII files, which reflect all of the essential information, are easily generated. ASCII files of this form are equally easy to import into FLOWforms. Since all tools provide the necessary information, it is a simple matter to import information from any other tool.

*Generating Text from Linked FLOWforms: The contents of individual FLOWforms either singly or collectively can be generated in a wide variety of ways. TEXT and SUBTREE FLOWforms, referenced in a higher level FLOWform, are automatically generated during code and report generation—just as if they were part of the original FLOWform. SUBTREE generation is recursive, thereby making it possible to create reports as large as desired while still maintaining modularity.\(^4\)*

---

\(^4\) The INCLUDED FILE mechanism also may be used recursively to optionally reference FLOWforms. This type of linking is most often used in code generation in such modern languages such as Pascal, C and Ada but can also be used with COBOL and FORTRAN. SUBTREE and TEXT linkages are especially convenient in working with reusable routines (or macros) in COBOL or FORTRAN.
Custom Extraction: PRODOC™ provides a flexible mechanism for extracting custom information from FLOWforms. One can attach labels to individual FLOW-form elements as desired (e.g., {DOS-specific, critical—timing}). Later, one can extract the contents of all nodes which satisfy any prescribed logical condition (e.g., “DOS-specific + CICS-call”).

Generating Calling Hierarchies and Cross Reference Reports: Hardcopy can also be used to generate calling hierarchies, indicating which FLOWform modules are used by other FLOWforms. Obviously, this is most useful in understanding the relationships which exist between FLOWforms.

The Find/Uses option in the Report Generator is used to generate cross reference listings. These reports tell which FLOWform modules use given identifiers and are convenient in determining the potential impact of anticipated system changes.

5 RE-ENGINEERING FROM A COGNITIVE PERSPECTIVE

With this background, let us reconsider re-engineering from a cognitive perspective or approach (CogApp). Rather than forcing a choice between redevelopment and salvaging old code, the CogApp to systems engineering and re-engineering puts new development and maintenance on the same plane. In this view, the choice is not either-or but how much new development and how much revision. The CogApp in either case involves knowledge engineering carried out at a relatively high level of abstraction, independently of how the code is actually structured.

Based as it is on a particular form of knowledge engineering, called structural analysis (e.g., see Scandura, 1982, 1984, 1987; Scandura et al., 1974), CogApp requires one to look at both data and process from the top down. From a cognitive perspective, it makes no sense to analyze data before considering process or process before data (although, of course, some kinds of systems involve relatively more data, and others more procedure). It is important at each stage of analysis to have a complete picture, or cognitive map, of the system at some level of abstraction. The CogApp calls for testing high level system designs independently of the existing or planned code.

Conversely, structural analysis assumes availability of certain basic capabilities that have meaning in the context of the desired system. If concerned with maintaining a business system, for example, one meaningful component might be making out an invoice. If the system were to control an airliner, then reasonable software components might detect air pressure or control flaps. In dealing with an existing system, this phase involves analyzing the code to determine whatever meaningful components might exist therein.

The CogApp involves three basic steps:

1) Create a high level or logical design which captures the functionality of the current or planned system. This may be done initially at a high level of abstraction as in developing any new system. Where the planned functionality of the system is not fundamentally different from the current functionality, one can typically move rather quickly to a high level design. To make this approach work it is highly desirable
that the design be tested empirically to see if functional expectations as to flow of
control (and/or displays) are as desired. (The basic approach is of the same genre as
those described by Boehm (1988) and Yeh (1990, Ch. 1) but with different emphasis.)

2) Reverse engineer existing code to extract the basic system design so it can more
easily be analyzed, evaluated, documented, restructured and/or otherwise modified.
The essential aspect is identifying meaningful components that might reasonably be
'cleaned up' and reused. Particular attention is paid to aspects of the code that
correspond to components of the high level design.

3) Map appropriately documented and modified low level components into the high
level design. Various adjustments such as renaming parameters or variables might
be made in the process but always with the goal of preserving the desired high level
design.

This approach has two important advantages:

a) Functionality of the revised system is reflected directly in the new higher level
design. This makes it relatively easy for both new and old programmers to modify
and otherwise maintain the code.

b) As much of the existing code as possible is retained (without distorting the desired
system) thereby reducing re-development costs—sometimes dramatically.

In order to efficiently implement the CogApp, we need a system development and
maintenance system like the re/NuSys Workbench. The re/NuSys Simulator provides
a way to develop and execute high level designs independently of the underlying
code. As noted above, testing designs at a high level of abstraction dramatically
reduces the number of necessary test items.

The re/NuSys Translator is used to automatically convert tested high level designs
into FLOWforms in the target language—C, C++, Ada, Pascal, Fortran or
COBOL.

The re/NuSys Reverse Engineer takes existing source code and reverse-enginners
it into hierarchical FLOWforms. In the process, messy notation such as 'BEGIN',
'END', '(', ')'; ':' ',' '.' is eliminated—leaving only the basic pseudocode that tells
humans what the system actually does.

After reverse engineered modules have been checked, the Simulator's higher order
semantic processing rules can be used where desired to recapture overall system
relationships (e.g., in the form of calling hierarchies).

The Designer's checking facilities are then used to semi-automatically create links
between the new high level design and the reusable, reverse engineered code. The
Designer also provides a highly responsive visual environment where needed FLOW-
form modules can more easily be created and existing modules modified.

FLOWform Summary

Key steps in the CogApp are summarized as FLOWforms in Figures 23–26. Figure
23 deals with design; Figure 24 with implementation; Figure 25 with re-engineering
and optional translation, and Figure 26, combining new designs with the reuse of
old code.
Specify and design system.

<table>
<thead>
<tr>
<th>REPEAT</th>
<th>Create/refine system specifications. [ScanFLOW Designer]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNTIL</td>
<td>Test system by executing specifications. [PRODOC Simulator]</td>
</tr>
</tbody>
</table>

IF more_rigorous_design_desired

THEN

<table>
<thead>
<tr>
<th>REPEAT</th>
<th>Convert/refine specifications to logical design. [ScanFLOW Designer]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>check pseudocode for syntax, consistency and redundancy.</td>
</tr>
<tr>
<td></td>
<td>Test system by executing logical design. [PRODOC Simulator]</td>
</tr>
<tr>
<td>UNTIL</td>
<td>design_satisfactory/tested</td>
</tr>
</tbody>
</table>

Figure 23

Implement Systems.

<table>
<thead>
<tr>
<th>CASE nature_of_system OF</th>
<th>Implement system in pseudocode in target language. [ScanFLOW Designer]</th>
</tr>
</thead>
<tbody>
<tr>
<td>straight_forward</td>
<td>Check pseudocode for syntax, consistency and redundancy.</td>
</tr>
<tr>
<td></td>
<td>Generate source code ready for compiler. [PRODOC Code Generator]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>cost&gt;efficiency</td>
<td>Implement system in PRODOC's 40L. [ScanFLOW Designer and Simulator]</td>
</tr>
<tr>
<td></td>
<td>IF adequate_in_speed_and_efficiency</td>
</tr>
<tr>
<td></td>
<td>THEN Generate full source code ready for compiler. [PRODOC Prototype Generator]</td>
</tr>
<tr>
<td></td>
<td>ELSE Introduce source code, implement or reuse additional routines. [ScanFLOW Designer and Prototype Generator]</td>
</tr>
<tr>
<td>max_efficiency</td>
<td>REPEAT IF reusable_routines_available</td>
</tr>
<tr>
<td></td>
<td>THEN Make contact with reusable routines. [ScanFLOW Designer and Simulator]</td>
</tr>
<tr>
<td></td>
<td>Translate logical design to target language. [PRODOC Translator]</td>
</tr>
<tr>
<td></td>
<td>Check FLOWSforms to create declarations in target language. [ScanFLOW Designer]</td>
</tr>
<tr>
<td></td>
<td>Generate full source code ready for compiler. [PRODOC Code Generator]</td>
</tr>
<tr>
<td></td>
<td>IF low_level_components_missing</td>
</tr>
<tr>
<td></td>
<td>THEN Implement components. [PRODOC in conjunction with &quot;deaniap&quot;]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNTIL satisfied</td>
</tr>
</tbody>
</table>

Figure 24
### Figure 25

**Reverse engineer existing source code to pseudocode in FL01forms.**

<table>
<thead>
<tr>
<th>CASE language to be maintained OF source</th>
<th>Add higher level annotation and restructure FL01forms as desired.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF code has linear structure(e.g., COBOL/4/FortranIV) THEN Modernize FL01form design hierarchies (e.g., to COBOL/5 or Fortran).</td>
<td></td>
</tr>
</tbody>
</table>

**CASE familiarity with language OF source**

<table>
<thead>
<tr>
<th>CASE familiarity with language OF source</th>
<th>Add higher level annotation and restructure FL01forms as desired.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF code has linear structure(e.g., COBOL/4/FortranIV) THEN Modernize FL01form design hierarchies (e.g., to COBOL/5 or Fortran).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>target</th>
<th>Convert to FL01forms in target pseudocode language.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>target</th>
<th>Add higher level annotation and restructure FL01forms as desired.</th>
</tr>
</thead>
</table>

---

### Figure 26

**Renew design with reuse of existing source code.**

<table>
<thead>
<tr>
<th>Re-engineer existing source code into FL01forms. [PRODOC in conjunction with &quot;Reeng&quot;]</th>
<th>Set reusability standards and assess FL01forms (including overall design) for reusability. [ScanFL01 Designer and Report Generator]</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF overall design inadequate(e.g., &quot;spaghetti code&quot;) THEN REPEAT</td>
<td>Create specifications and logical design to reflect overall system functionality. [ScanFL01 Designer]</td>
</tr>
<tr>
<td>Test by executing specifications and logical design. [PRODOC Simulator]</td>
<td>Adjust designs as necessary to make contact with reusable FL01form modules and/or macros. [ScanFL01 Designer]</td>
</tr>
<tr>
<td>UNTIL satisfied</td>
<td>Check FL01forms to create links with reusable FL01form modules. [ScanFL01 Designer]</td>
</tr>
<tr>
<td>Translate logical design to target language (e.g., COBOL). [PRODOC Translator]</td>
<td>Check FL01forms to create declarations in target language (e.g., COBOL). [ScanFL01 Designer]</td>
</tr>
<tr>
<td>IF low level components missing</td>
<td>Implement components. [PRODOC in conjunction with &quot;desnisp&quot;]</td>
</tr>
</tbody>
</table>

---
References


