Complexity Measurement of a Graphical Programming Language and Comparison of a Graphical and a Textual Design Language

by

Roger Allen Goff

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Dr. Sallie M. Henry, Chairman

Dr. H. Rex Hartson

Dr. Dennis G. Kafura

Blacksburg, Virginia

Complexity Measurement of a Graphical Programming Language and Comparison of a Graphical and a Textual Design Language

by

Roger Allen Goff

Dr. Sallie M. Henry, Chairman

Computer Science and Applications

(ABSTRACT)

For many years the software engineering community has been attacking the software reliability problem on two fronts. First via design methodologies, languages and tools as a precheck on quality and second by measuring the quality of produced software as a postcheck. This research attempts to unify the approach to creating reliable software by providing the ability to measure the quality of a design prior to its implementation. Also presented is a comparison of a graphical and a textual design language in an effort to support cognitive science research findings that the human brain works more effectively in images than in text.

Acknowledgements

I would like to thank my advisor, Sallie Henry, for her support and guidance in this research. Her quick solutions to problems in times of crisis helped to keep my sanity. I would also like to thank Jeff Vining in the statistics consulting lab who advised me not only on what statistical analysis I should perform, but on how to interpret the results and Rex Hartson and Dennis Kafura for serving on my committee.

A special thanks goes out to whose initial work at debugging the analyzer and preprocessing of data was a great help. He was also there for me to complain to when things weren't going right. Also I would like to thank for his friendship and companionship throughout my graduate education and especially during the first year.

Last, but certainly not least, I would like to thank my family, , and friends for their support throughout my entire education without which many things would have been more difficult than they were.

Table of Contents

List of Illustrations

List of Tables

Chapter 1. Introduction

Introduction

The software design phase can generally be divided into two major stages, namely architectural or general design, and detailed design [YAUS 86]. General design is responsible for translating requirement specifications into system structure. This stage places emphasis on module definition and on the interrelationships among modules. Design techniques include the concepts of top-down design and functional decomposition. The system architect begins by analyzing the requirements specification and decomposing the system into functional units or modules. The major concerns are not how a problem is to be solved, but instead, differentiating the sub-problems of the problem and documenting the communication which must occur among the sub-problems. The process of identifying the sub-problems of a problem is commonly referred to as functional decomposition and its recursive application within the sub-problems defines the system structure and leads to the detailed design stage where the architectural design is transformed into the procedural description of a software system. This second stage or detailed design phase emphasizes the selection and evaluation of algorithms. Design techniques for this level include the concepts of bottom-up design and stepwise refinement. Once the structure of a system is defined to a point where the solution to the sub-problem becomes obvious, a designer may discontinue the top-down design approach

and begin designing bottom-up via the definition of utility routines to be used in the solution. The solution to the sub-problem is then defined by application of these utilities within specific routines. Some designers never revert to designing in a bottom-up fashion, instead, they continue the top-down approach by laying out the algorithmic structure within procedures while avoiding actual representation details as long as possible. The continuance of top-down design is called stepwise refinement. The two stage definition of design aides in the discussion of the different design methodologies previously and currently employed.

The detailed design phase of the software life cycle is the focus of this thesis. With this in mind, tools to assist the designer with this phase are briefly described.

Flowcharts

Flowcharting is the traditional means for specifying and documenting the algorithmic details in a software system and over the years a great controversy has developed concerning its usefulness [FAIR 85]. Flowcharts have been called everything from "an essential tool in problem solving" [BOHM 71] to "an obsolete nuisance" [BROF 75]. Flowcharts are generally accepted as a useful tool for diagramming program flow of control, however, problems arise when a flowchart gets large and no longer fits on a single page. Other weaknesses include the inability to represent complex decision predicates legibly in the small space of a decision diamond and the frustration associated with drawing flowcharts and having no easy way to modify them. Therefore other representations evolved.

HIPO Diagrams

A derivative of flowcharts called HIPO diagrams (Hierarchy-Process-Input-Output) were developed at IBM as " design representation schemes for top-down software development, and as external documentation aids for released products" [FAIR 85]. HIPO charts consist of three parts; a visual table of contents, a set of overview diagrams, and a set of detail diagrams. The table of contents is a tree structured or graph structured directory of the contents of the overview diagrams. The table of contents also includes a legend of symbol definitions. The overview diagrams specify the inputs, processing steps and outputs of a functional unit of the system. The overview diagram format consists of boxes that are filled with English-like phrases describing the execution process and connected with arrows to indicate the flow of control. The internals of the boxes may also contain a description of the procedure interface, local data structures, timing constraints, side effects, and procedure limitations. Detail diagrams have the same format as overview diagrams. HIPO charts are an attempt to bypass the obvious problems found with flowcharts, and are a step in the right direction in terms of the elimination of off-page connections and the use of English-like phrases for decision predicate specification. However, the inability to easily modify the charts still exists.

Pseudocode

Another widely used design tool is pseudocode. Like flowcharts and HIPO diagrams, pseudocode can be used to represent both detailed and architectural designs. Using pseudocode, the designer uses english-like language phrases intermixed with keywords like WHILE, and IF-THEN-ELSE. The use of the keywords combined with indentation shows the flow of control within the system, and the english phrases describe the processing. Since there are no structuring rules, each individual uses his own style and syntax. Upon examination of several designers' concepts of pseudocode format, the need for some sort of standardization becomes apparent.

PDL

By adding syntactic requirements to pseudocode, standardization of pseudocode format is achieved and one defines what is called a program design language or PDL. Program design languages can be used in both the architectural and detail design stages. A good PDL is adaptable to its environment, is familiar to both designers and programmers, and has a nonrestrictive syntax to allow the designer to freely express himself. Among the advantages of PDL's is the ease of modification, since they can be stored in text files. Other advantages are the ability to syntactically check the design and easily translate a design to high level code. A study comparing PDL's to flowcharts found PDL's to be a superior design tool [RAMH 83]. PDL's, however, have some disadvantages not found in flowcharting. Like high level programs, PDL's do not provide a clear representation of the flow of control in a system, and their use requires the burden of an imposed syntax.

GDL

Psychological studies have shown that the brain processes visual information faster than verbal information and that humans actually think in pictures. These findings are the motivation for designing with graphs. Bill Benzon, a researcher in cognitive science, has stated that, "if you want to think about how ideas fit together and to teach this material to others then graph notation is the most useful" [BENB 85]. Graphs permit the designer to visualize all the components of a system quickly and easily, thus allowing the thought process to concentrate on development. A textual listing of the same system requires the designer to read each line and painstakingly assemble a view of the system and its components. Design graphs are also easily understandable by inexperienced programmers and nonprogrammer managers since there is little syntax to learn and only a small set of symbols with which to become familiar. Graphical design languages are excellent tools for architectural design, and in contrast to their graphical ancestors are well suited to detailed design.

GDL's can also have the added feature of automated translation to high level programming language code. Versatility and ease of use are responsible for the wider acceptance and application of GDL's in real world environments.

Existing GDL's

Progam Visualization

The Program Visualization environment, which was developed at the Computer Corporation of America, is devoted primarily to the dynamic graphical execution of systems. The graphical system "simulation", is concerned mainly with the changes made to structured data. The developers maintain that the depictions of flow of control do not need to be as complex as the depictions of data [BROG 85]. Program Visualization uses multiple windows to display both code and graphics while showing control flow within the windows by highlighting the portions of the system being executed The graphical depiction of data has the capability to monitor and display changes in such common data structures as numeric variables, arrays, linked lists, and trees. The user can create graphical pictures and link them to code and data, thus establishing high-level icons that can be used during execution monitoring. The level of abstraction displayed can be chosen by the programmer, with choices ranging from top-level drawings of the whole system being developed to bottom-level depictions of language code. The construction of programs and static graphics by assembly can be viewed as an extension of current practices, in particular, the use of subroutine libraries. The Program Visualization system provides a navigational aid facility for access to the library routines. Typically the library would contain both graphical and code templates for commonly used data structures and graphical symbols or building blocks. Kits are mixed collections of building blocks and templates. An example of the use of the library might be the storage of standard flowchart notation in a kit. The user can then create flowcharts by simply copying symbols from the kit into the diagram window, filling in the text as is necessary.

PegaSys

PegaSys is an experimental system whose main purpose is to use graphical images as formal, machine processable, documentation and an explanation of system designs. A program design is described in PegaSys by a hierarchy of interrelated pictures. Each picture describes data and control dependencies among such entities as "subprograms," "processes," "modules," and "data objects" [MORM 85]. Particularly interesting is the PegaSys system's ability to: check whether pictures are syntactically meaningful, enforce a design development methodology throughout its functional decomposition, and determine whether a resulting program meets its pictorial documentation. These abilities are achieved through the use of a graphical calculus formulae representation of diagrams. A grammar for the graphical calculus allows for syntactic checking and for enforcement of the decomposition process. PegaSys also provides a full featured graphical editing system for the creation and refinement (functional decomposition) of diagrams.

Pict

The Pict system is a departure from the approach of the preceding systems in that it is designed to aid program implementation rather than algorithm design and selection. Pict is actually a program development environment, providing all the tools users need to compose, edit, and run their programs while being integrated within a simple consistent command structure [GLIE 84]. Once Pict is entered, the user need not touch the keyboard; communication with the system is by some suitable input device like a joystick or mouse. Programs are constructed using graphical icons to represent data structures, parameter passing modes, variables, and program operations. Even subprogram names are represented by icons that are created by the user with an icon editing facility. Control structures are represented by colored directed paths. Program syntax is continuously checked during development, and inconsistencies are eliminated --- even when a program or subprogram is only partially specified. Once a subprogram is developed, the icon representing its name

is inserted into a menu tree, this icon can then be accessed to reuse the subprogram in other developments. A user may also delete an icon from the menu tree, and all links to that icon are automatically removed.

GPL

GPL is the graphical programming language of the Dialogue Management System, DMS, being developed at Virginia Tech [HARH 85]. GPL follows the Supervised Flow design methodology which dictates that each program and subprogram has a supervisor which supervises all the flow of information within a diagram. The diagrams of the programs are called Supervised Flow Diagrams, or SFDs. The DMS environment provides a graphical editor for the creation of SFD's which consist of a small set of icons representing decisions, subprogram calls, input/output operations, statement blocks, start, return, and control and data flow lines. GPL, like Pict, is unique in the sense that it too is a programming language rather than just a design language. The DMS system has a coder, which takes the SFD's for a program and generates high level language code. Currently the coder generates 'C' code, however, eventually a user will have a choice of languages to generate. The DMS project also has plans for a Behavioral Demonstrator which will dynamically execute a program, while allowing the user to examine and change variable values, and exhaustively test parts of a program [CALJ 85], [SMIE 86].

Conclusion

Having recognized the need for a structured design methodology, many different design tools and techniques developed including: flowcharts, HIPO diagrams, pseudocode, program design languages and graphical design languages. The transition to each new technique is motivated by the inadequacies of currently available techniques. The most recent transition from PDL's to GDL's

has sound theoretical support from the field of cognitive science which concludes that the human brain processes pictorial information more efficiently. In this research an automated software quality metric analysis tool is used to verify that the complexity of graphical designs can be measured and, more importantly, that programs designed using a graphical design language are less complex than those designed with a PDL.

Chapter two gives a brief description of some of the established software quality metrics along with a description of the software quality metric analysis tool used in this study. In chapter three is a detailed description of the graphical design language, used in this study, and a technique for analyzing the language. The results of measuring the graphical language are presented in chapter four and a comparison of the graphical language and a textual design language is given in chapter five. Chapter six contains an intermetric analysis that supports the findings of an earlier metric study and a description of some earlier results. Finally, chapter seven recounts conclusions and identifies future work.

Chapter 2. Overview of Metrics

Introduction

While some of the software engineering community has been dedicating time to developing tools and methodologies to aid in the design process, another sector of the community has been developing the means to measure and analyze the quality of software. An enormous amount of literature has been written covering a broad spectrum of proposed measurements. For this study the qualitative, nonautomatable and nonintuitive metrics are ignored. Nonautomatable or qualitative metrics are not of interest to the practitioners. Measuring a real system of 500,000 lines of code is not practical unless the measuring technique has been automated. In this chapter some of the more significant automatable metrics are described along with an automated software quality metric analysis tool.

Three classifications of automatable metrics to be examined are micro, or code metrics, macro, or structure metrics and hybrid metrics. The micro metrics are concerned with measuring features taken directly from source code, for example, length of a program and the number of decision constructs in program are considered micro metrics. Macro metrics, however, are concerned with measuring the complexity of the structure of a software system, for example, measuring the com-

plexity of the decision process that leads to the invocation of a module. Finally, hybrid metrics are those metrics that utilize a combination of code and structure metrics to produce complexity measures.

Micro Metrics

Length

The most familiar software measurement is lines of code, or length [CONS 86]. This may seem to be a simple metric that can be easily derived, yet there is no general agreement as to what constitutes a line of code. Conte, Dunsmore and Shen [CONS 86] propose the following definition:

line of code A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all line containing program headers, declarations, and executable and non-executable statements

Most researchers agree that the line of code measure should not include comments or blank lines since their presence or absence does not affect the functionality of a program. Interpretation of the above definition for various source languages could include such counts as the number of semicolons, in Pascal or the number of carriage returns in 'C'.

McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity [MCCT 76] denoted $V(G)$, has its foundations in graph theory. This code metric measures the number of basic paths in a routine, and is derived from the representation of a procedure as a strongly connected graph. The nodes in the graph represent blocks

of code in the procedure where the processing is sequential and the arcs correspond to branches taken. Each program graph must have a unique entry and exit point. In order to perform the complexity analysis, a procedure's control graph must be strongly connected, a single arc from the unique exit point is drawn back to the unique entry point to satisfy this criterion.

A result from graph theory states that in a strongly connected graph G , the cyclomatic number, $V(G)$, is equal to the maximum number of linearly independent circuits, where

$$V(G) = E - N + 2 \quad \text{and where}$$

E = the number of edges in the graph

N = the number of nodes in the graph

In this way McCabe defines the cyclomatic complexity, $V(G)$, of a procedure to be the cyclomatic number of the procedure's strongly connected control graph.

The reasoning behind McCabe's definition is that the complexity of a procedure is going to be proportional to the cyclomatic number or, alternatively, the number of linearly independent circuits in the procedure's control graph.

The complexity of a program of p procedures would be the sum of the complexities of each of the p procedures and is given by:

$$\begin{aligned} V(G) &= \sum_{i=1}^{p} V(G_i) \\ &= \sum_{i=1}^{p} (e_i - n_i + 2) \\ &= e - n + 2p \end{aligned}$$

where,

G_i = procedure i's graph

e_i = the number of edges in procedure i's graph

n_i = the number of nodes in procedure i's graph

e = the total number of edges in all graphs

n = the total number of nodes in all graphs

For structured programs McCabe simplified the calculation of the cyclomatic complexity to a count of the number of simple predicates in a routine plus one utilizing results derived by Mills [MILH 72].

Halstead's Software Science

Halstead's code metrics [HALM 77] take the view that algorithms have physically measurable characteristics. The resulting theory, called Software Science, states that certain algorithmic properties may be derived from the following four counts:

$n1$ = the number of unique operators appearing in a program

$n2$ = the number of unique operands appearing in a program

$N1$ = the total number of occurrences of all operators in a program

$N2$ = the total number of occurrences of all operands in a program

Two definitions, derived from the above counts, are used in the calculation of the Software Science metrics. The sum of the counts $n1$ and $n2$, define the size of the vocabulary of a program and is denoted by n. The length of a program, denoted N, is similarly defined as the sum of $N1$ and $N2$.

In the following sections three Software Science metrics, their derivation, and intuitive meanings will be described.

Program Volume

An important algorithmic characteristic is size. The following formula estimates the size of an algorithm based on the size of the algorithm's vocabulary and length:

$$V = N \times \log_2(n)$$

The derivation of this formula is subject to two interpretations. The first treats V as a measure of the number of bits required to store the algorithm since it takes a minimum $\log_2(n)$ bits to represent each of the symbols in the vocabulary. The second interpretation treats V as the number of mental comparisons required to generate the algorithm. The second interpretation assumes that a programmer must make a binary search through the vocabulary each time an element in the algorithm is selected.

It is important to note that the program volume metric's value will vary in size, based on the expressive power of the algorithm's implementation language. Similarly, in any one language some algorithms are smaller in size than others. These features are required of the volume metric in order that such changes may be studied.

Program Level

The program level metric is defined as the ratio of the potential volume , V^*, to the actual volume, V, of the algorithm:

$$L = \frac{V^*}{V}$$

where the potential volume is the most succinct form in which an algorithm could ever by expressed. Since the value of V^* cannot be established, it is necessary to derive an estimator in terms of operands and operators.

The least number of operators that could ever be employed is two, where the pair would be a function call and an assignment operator. However, there appears to be no limit to the number of operators that one might employ. From this we have the ratio:

$$L \sim \frac{2}{n1}$$

with respect to operators.

On the other hand, operands exhibit no unique minimum value, so they must be treated on an intuitive level. Whenever an operand is repeated, it is an indication that the algorithm's implementation is at a lower level. This effect may be demonstrated by the ratio:

$$L \sim \frac{n_2}{N_2}$$

Combining these two ratios gives us the program level metric's estimator:

$$\hat{L} = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

A second interpretation of the program level measurement is the relative intellectual ease with which a programmer makes a single mental comparison.

Programming Effort

The programming effort metric can be thought of as a measure of the mental activity required to translate a collection of ideas regarding a problem solution into an actual implementation of the algorithm in some language known to the programmer. The formula for the effort metric is:

$$E = V \times D$$

where V is the program volume metric, and D is the reciprocal of the program level metric L. D represents the difficulty of making the mental comparisons required to implement the algorithm and is measured in units referred to as elementary mental discriminations (EMD). Replacing D in the equation yields:

$$E = \frac{V}{L}$$

By substituting in L's theoretical value we derive another equation from which significant conclusions can be cited:

$$E = \frac{V^2}{V^*}$$

This equation implies that the mental effort required to implement any algorithm varies by the square of its volume, not linearly, and therefore that proper modularization of algorithms can definitely reduce programming effort.

Structure Metrics

McClure's Module Invocation Complexity

McClure's structure metric [MCCC 78] examines two characteristics of a module in order to define a complexity. The first characteristic is a measure of the complexity of the circumstances under which the module was invoked and the second is a measure of the complexity of of a module invoking another module.

The complexity of invoking a module is dependent on two factors, and requires the definition of a module's invocation control variable set. The invocation control variable set for a module is the

set of variables whose values effect a particular invocation of a module. For example, suppose the following code is used to invoke module B:

```
while (var1 < var2) or (var3 = 0) do
        if (var4 = var5) then
            invoke
end  while;
```

The set $\{var1, var2, var3, var4, var5\}$, forms an invocation control variable set for module B. It is possible for a module to have more than invocation control variable set if it is invoked conditionally in more than one place. Given this definition, the complexity of the invocation of a module is a function of the number modules that invoke the module, and the average invocation control variable set complexity where the complexity of an invocation control variable set is given by:

$$b \times \sum_{i=1}^{e} C(V_i)$$

with,

b	=	2 if any V_i is a control variable of a repetition structure
	=	1 otherwise
$C(V_i)$	=	Control Variable Complexity [MCCC 78]
e	=	the number of variables in the invocation control variable set

The complexity of a module invoking another module is a function of the number of modules which it invokes and the average complexity of all invocation control variable sets used by the module in invoking its direct descendants.

Combining the above two characteristics gives the following formula for the complexity of a module B:

$$M(B) = [F_p \times X(p)] + [G_p \times Y(p)]$$

where,

F_p = the number of modules that invoke B.

$X(p)$ = the average complexity of all invocation control variable sets used to invoke module B.

G_p = the number of modules invoked by module B.

$Y(p)$ = the average complexity of all invocation control variable sets used by module B to invoke other modules.

Henry and Kafura's Information Flow Metric

Henry and Kafura [HENS 79], [HENS 81] have developed a structure metric which derives a procedure's complexity from the procedure's connections with its environment. These environment connections are taken from the flows of information in and out of the routine, termed fan-ins and fan-outs respectively. The formal definitions of fan-in and fan-out are as follows:

fan-in the number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information

fan-out the number of local flows from a procedure plus the number of global data structures which the procedure updates.

The fan-ins and fan-outs for a routine are determined in a two step process. The first step involves the generation of a set of relations which represent the flow of information within a routine via input and output parameters, updates of global data structures, reads from global data structures and returns from function calls. From these relations a forest of trees is built which has as its roots global data structures. The leaves represent data structures and function returns whose information will ultimately end up in the global data structure at the root of their tree. The internal nodes represent interfaces through which the information at the leaves will travel on their way to the root.

Thus, each tree represents all possible paths of information flow into a particular global data structure.

The formula defining the complexity value of a procedure is given by:

$$C_p = (fan\text{-}in \times fan\text{-}out)^2$$

where,

C_p	=	the complexity of procedure p.
fan-in	=	the number of fan-ins to procedure p.
fan-out	=	the number of fan-outs from procedure p.

The term (*fan-in* × *fan-out*) represents the total possible number of combinations of an input source and an output destination, and is raised to the power of 2 similarly with Brook's law of programmer interaction [BROF 75] and Belady's formula for system partitioning [BELL 76].

Two other related complexities are defined, a module complexity and a level complexity. The module complexity with respect to a global data structure is the sum of the procedure complexities of those procedures that retrieve information from or update information to the data structure. When viewing a system as a hierarchy of levels, the complexity of a level is defined to be the sum of the complexities of the modules which comprise that level.

Hybrid Metrics

Woodfield's Syntactic Interconnection Model

Woodfield's Syntactic Interconnection Model [WOOS 80] is a hybrid metric which attempts to measure programming effort in terms of time. The general equation of the model is given by:

$$C_B = C_{iB} \times \sum_{k=2}^{fan\_in_i - 1} RC^{k-1}$$

where,

C_B	=	the complexity of module B's code
C_{iB}	=	the internal complexity of module B's code
fan_in_i	=	the sum of the control and data connections for module B's code
RC	=	a review constant

It is important to note that the internal complexity for a module may be any code metric, however, in Woodfield's model definition Halstead's Program Effort metric was used.

The model uses a review constant of $\frac{2}{3}$, which is a number previously suggested by Halstead [HALM 77]. Woodfield completes his model by defining the fan_in of a module to be the combination of control and data connections for the module. A control connection exists for a module each time it is invoked, while two modules, A and B are data connected if there is some variable V such that the following two conditions are true:

1. the variable V is modified in B and referenced in A

2. there exists at least one data path set $D(BA)$, between B and A, such that the same variable V is not referenced in any d_i where d_i is a member of $D(BA)$

where a data path set, $D(BA)$ is an ordered set of one or more modules $\{d_1, d_2, d_3, ..., d_n\}$ such that one of the following conditions is true:

1. B calls D_1; D_1 calls D_2; ... ; D_{n-1} calls D_n; D_n calls A.

2. A calls D_1; D_1 calls D_2; ... ; D_{n-1} calls D_n; D_n calls B.

3. There exists some D_i in $D(AB)$ such that D_i calls both D_{i-1} and D_{i+1} ; D_{i-1} calls D_{i-2}; D_{i-2} calls D_{i-3}; ...; D_2 calls D_1; D_1 calls B; and also D_{i+1} calls D_{i+2} ; D_{i+2} calls D_{i+3}; ... ; D_{n-1} calls D_n; D_n calls A.

Henry and Kafura's Information Flow Metric

Henry and Kafura's Information Flow metric may also be applied as a hybrid metric, in fact, the metric has been tested in its hybrid form on the UNIX operating system [HENS 81], [KAFD 85]. The formula for the Information Flow metric in its hybrid form is as follows:

$$C_p = C_{ip} \times (fan\text{-}in \times fan\text{-}out)^2$$

where,

C_p	=	the complexity of procedure p.
C_{ip}	=	the internal complexity of procedure p.
fan-in	=	the number of fan-ins to procedure p.
fan-out	=	the number of fan-outs from procedure p.

Note that C_{ip} may be any micro metric's measure of procedure p's complexity. In this study the Information Flow metric is combined with the length, effort and cyclomatic complexity measures.

The Software Quality Metric Analysis Tool

The software quality metric analysis tool, subsequently referred to as the analyzer, used in this study has been developed over the last three years, at Virginia Tech, under the direction of Dr. Sallie Henry and Dr. Dennis Kafura. The development of the analyzer has been an ambitious project and preliminary results of its use are encouraging.

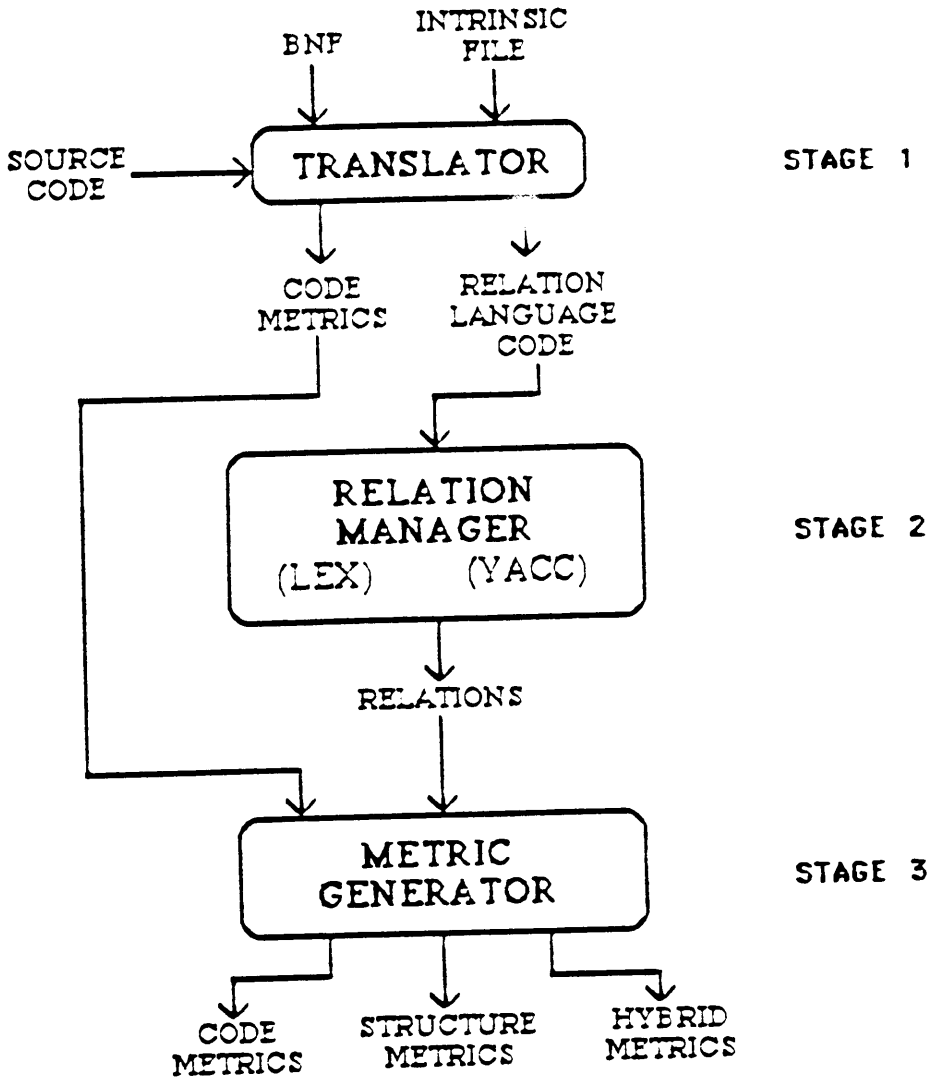


Figure 1. Logical Structure of the Software Quality Metric Analyzer

The analyzer takes source code as input, and calculates the values of the metrics discussed earlier in this chapter. The first two stages process source code one procedure at a time, while the third stage requires the information files from the entire system. A translation stage, a relation processing stage and a metric generation stage comprise the transition from source code to metric values. Figure 1 on page 21 shows the logical structure of the analyzer. The three stage approach to complexity analysis is motivated by the syntactic and semantic differences of analyzeable languages. Each source language to be analyzed must have its own code processor and translator, however, there is only one relation processor.

The translation stage of the analyzer is the only language dependent stage involved in the metric analysis. This stage simultaneously performs two functions. The first function involves the calculation of micro metric values for each procedure. The micro metric values are output to a file that serves as input to the metric generation stage. The second function of the translation stage translates the source code to an intermediate language, the relation language [HENS 88]. The relation language code preserves control and information flow while hiding proprietary information. The translation stage of the analyzer has been implemented for the Pascal, 'C', FORTRAN, and THLL programming languages.

The relation processing stage of the analyzer takes the relation language code, produced in the translation stage, and outputs information flow relations [KAFD 82]. The information flow relations serve as input to the metric generation stage.

The metric generation stage, or "back end", of the analyzer has as input the two files, micro metric file from stage one and the relations from stage two, and provides, via a menu driven interface, the means to run, examine and output each of the previously mentioned micro, structure and hybrid metrics. Module and level definition, analysis and output facilities are also provided.

Conclusion

The software quality measurement and analysis sector of the software engineering community has made great strides in the analysis of real time software systems. Studies have shown that many of the metrics developed do in fact identify stress points and areas with high error counts in existing systems. Language independence and the ability to identify different types of complexities in software have also been demonstrated. However, until recently, few attempts have been made to unite the processes of design and analysis. This unification is necessary to provide software engineers the ability to create reliable software systems. This study is one of two attempts to provide this unification. The other study focuses on use of a textual design language [SELC 87].

Chapter 3. Graphical Language Translation

Introduction

This Chapter describes the translation of the graphical language to its analyzeable form. Translation of GPL is a twofold process. First the source code is analyzed to generate code metrics used by stage three of the analyzer. The second process is the generation of relation language code while preserving the control and information flow of the corresponding source code. A detailed description of the graphical language, the relation language and the actual translation process is presented.

GPL

Introduction

The Dialogue Management System is a software system development environment being developed at Virginia Tech. Its underlying methodology is called SUPERvisory Methodology and Notation,

or SUPERMAN [HARH 85]. One of SUPERMAN's primary targets in software design is the separation of dialogue, communication between a user and the system, and computational design. This separation of system design is motivated by the impression that the best dialogue designers are not necessarily programmers and vice versa. The Dialogue Management System provides a graphical programming language, GPL, with which each type of designer may specify his design. GPL has a set of symbols specifically used for dialogue design and a set of symbols used for computational design with some symbols appearing in both dialogue and computational designs. These "linking" symbols tie the two parts of a system's design together. In this study, the computational development environment and the symbols in GPL used to create the computational portion of a system's design are of interest.

The basic building block for a GPL computational design is a Supervisory Cell which contains a supervisor and a Supervised Flow Diagram, or SFD. The SFD shows the flow of control and information within the cell. Intuitively, a supervisory cell represents the definition of a single subroutine in a system. Figure 2 on page 26 presents the symbols in GPL used for computational design and Figure 3 on page 27 presents a simple example of a supervisory cell. A nontrivial sample design appears in Appendix A with the corresponding source code located in Appendix C. The following sections discuss the syntax and semantics of each symbol and the information associated with each supervisor.

The Computational Design Symbols in a SFD

Control Flow Arcs

Control flow arcs show the flow of control throughout an SFD. These arcs can connect any two symbols of GPL together with the exception of databoxes, defined below. The arc labeled $x = 1$ in Figure 3 on page 27 is a control flow arc. Control flow arcs may or may not have a conditional associated with them. If there is only one control flow arc leaving a symbol there is no conditional

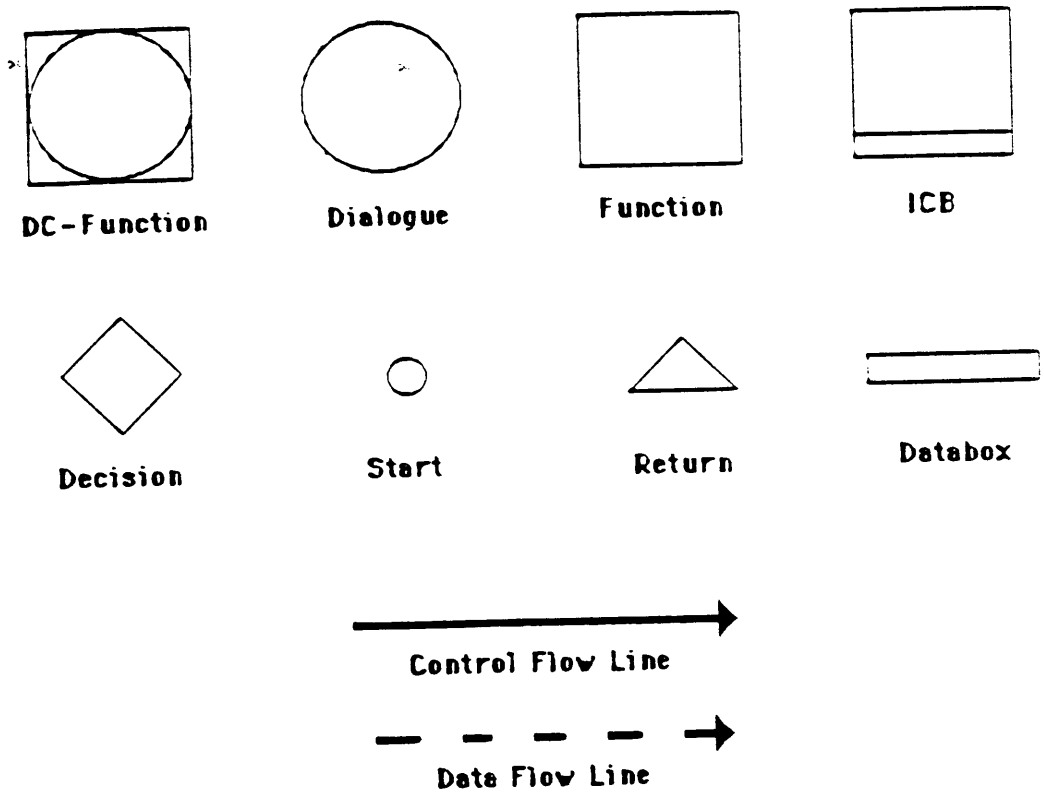


Figure 2. Computational Design Symbols in GPL

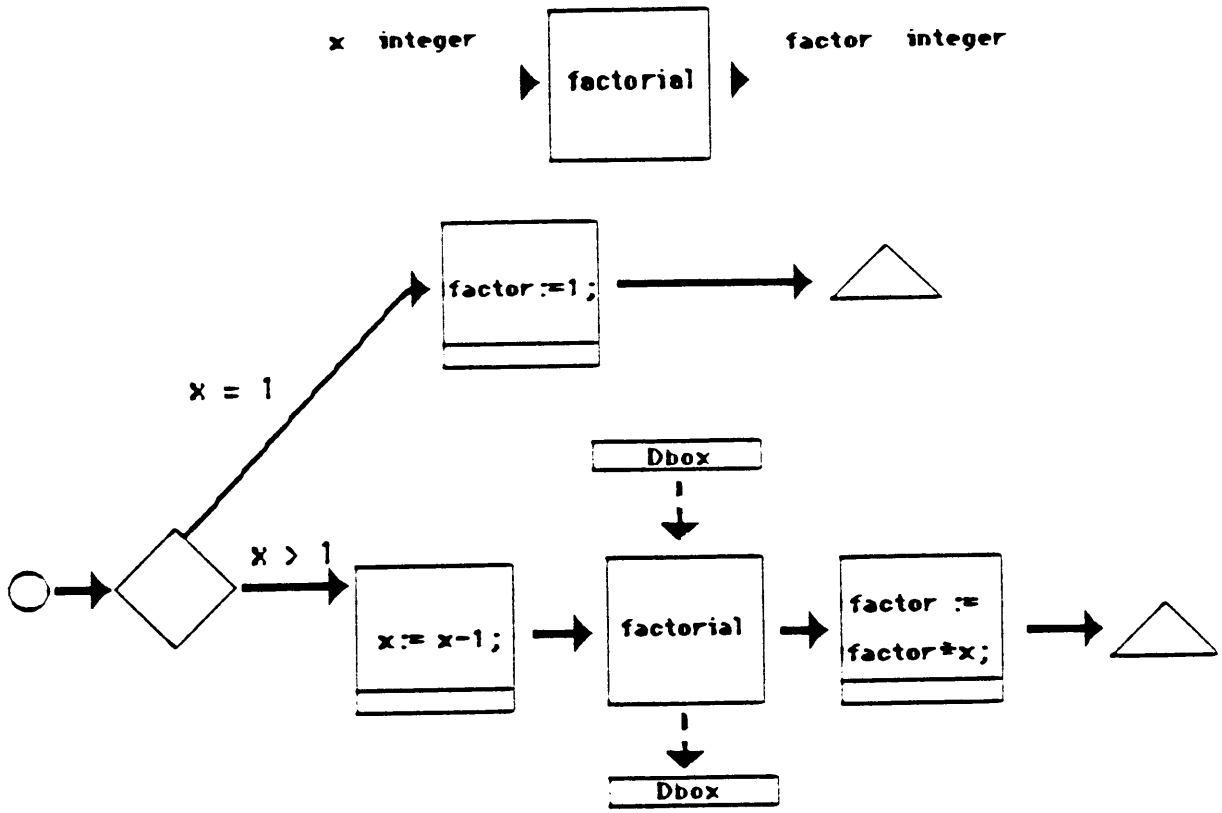


Figure 3. A GPL Supervisory Cell

on the arc, however, if there is more than one arc leaving a symbol there must be a conditional associated with each arc that leaves the symbol. A conditional associated with an arc is a valid boolean expression.

Data Flow Arcs

Data flow arcs are used to bind databoxes, defined below, to functions, also defined below. There can only be one data arc connecting a subroutine and a databox, and there is never a condition associated with a data arc. The dashed arcs leading into and out of the box labeled *factorial* in Figure 3 on page 27 are data flow arcs.

Start

There is exactly one start symbol per SFD. The start symbol marks the beginning of the execution of an SFD. It has no arcs coming into it, yet, it may have any number of arcs leaving. The circle in Figure 3 on page 27 is a start symbol.

Return

There is at least one return symbol per SFD. Returns have no arcs out of them, but they may have any number of arcs entering them. The return symbol represents the termination of execution of an SFD. Returns have no parameters and therefore are not used to return variable values. The conclusion is that there are no functions that return values in GPL. The two triangles in Figure 3 on page 27 are return symbols.

Decisions

A decision symbol may have any number of arcs entering it and any number of arcs leaving it. Its semantics resemble those of the Pascal "case" and 'C' "switch" statements. Each arc that leaves a decision must have a boolean expression associated with it. There are a minimum of two arcs leaving a decision symbol. The diamond in Figure 3 on page 27 is a decision symbol.

Databoxes

GPL databoxes are used to specify the actual input and output parameters to a subroutine call. They are sometimes called binding boxes since their content consists of the actual parameters and the names of their respective formal parameters. Databoxes have either a data flow arc leading into the box, i.e. an output parameter, or out of the box to identify an input parameter. The two rectangles labeled *Dbox* in Figure 3 on page 27 are databoxes.

Inner Code Block (ICB)

An ICB is a symbol that contains the actual code of a system. The code is syntactically and semantically correct high level language code. In theory, all of a program's code could be in an ICB, however, this use of an ICB is not intended. An ICB in a completely refined design contains only assignment statements. Any number of arcs may enter and leave an ICB. The symbol labeled $x := x - 1$; in Figure 3 on page 27 is an ICB.

Functions

The function symbol represents a call to a subroutine that contains no dialogue. A GPL function is unlike a Pascal function since it does not return a value. Each function symbol contains a name used to identify the corresponding supervisor for that function's definition. Functions may have any number of arcs leading into and out of them. Any time a function has more that one arc leaving it, each of the arcs must contain a conditional indicating the conditions necessary to follow that arc. If the function has input and output parameters in its definition, then there must be input and output databoxes attached, via data flow arcs. If the function does not have both input and output parameters, it is only necessary to have the appropriate databox (or none). Function definitions are not nested in GPL, therefore, references to the same function name in different SFDs refer to the same function definition. The box labeled *factorial* in the SFD in Figure 3 on page 27 is a GPL function.

Dialogue

Dialogue symbols represent input and output operations and may have any number of arcs leading to them and leaving from them. In the Dialogue Management System, this is where the dialogue designer, or author, takes over the design process. Computational designers either get information from a dialogue function, input, or give information to a dialogue function, output and are not concerned with how the information is manipulated. In the development of an SFD, a computational designer does not expand or further develop dialogue functions. Dialogue symbols, like functions, may have databoxes attached to them, via data flow arcs. There are no dialogue symbols in Figure 3 on page 27.

DC-Functions

DC-Functions, or Dialogue-Computation Functions, represent subroutines that contain both dialogue and computational operations. Aside from containing calls to dialogue functions, DC-Functions have the same syntax, semantics and requirements as functions. There are no DC-Functions in Figure 3 on page 27.

The Supervisor

There is only one supervisor per SFD, and it appears at the top of the diagram. Associated with the supervisor is a list of input parameters, a list of output parameters, a list of local variables, a name, and an SFD. When defining a supervisor's parameters and local variables, each < ID > given has a named type associated with it, however, as far as GPL is concerned there are no meaningful types or ways to define them. Therefore the types specified, are meant to be meaningful only to the designer and programmer. The box at the top of Figure 3 on page 27 labeled *factorial* is a supervisor.

Observations

GPL supports many desirable concepts of software engineering. First, there is no means of defining global variables. To some this may appear as a disadvantage, however, eliminating the ability to define a global variable is an excellent means to control its use. Secondly, and more importantly, by limiting the size of the work area, the definition of shorter more modular routines becomes natural.

GPL has some limitations that are particularly annoying. First is the inability to define system constants, needed for readability purposes and second, the inability to define libraries of routines for the purpose of reusability.

The Relation Language

Introduction

The relation language [HENS 88] was developed to address one of the major problems facing software engineering researchers, acquiring "real world" data to perform validation experiments. One good source of data, software organizations, is unavailable due to the proprietary nature of the products. Software organizations do not want their algorithms or the error history of their products released into public domain. With this in mind, the relation language was designed to serve as an intermediate language that can protect the software organizations' right to privacy while providing sufficient information for software engineers to test their measurements. In this discussion a brief description of the relation language constructs, presented in Figure 4 on page 33, is given.

Relation Language Constructs

Variables

A relation language program allows for the declaration of three types of variables: *local* , *struct* and *const*. The variable type one chooses is dependent upon the usage of the variable in the original source code. Distinctions are made between data objects as being local variables, complex structures and constants. Variables defined as type *local* are considered local to the routine in which they are declared. Determination of local variables is not always a simple process due to the presence

Declaration statements

LOCAL	-	local variable declaration
STRUCT	-	non-local variable declaration
CONST	-	defined constant
EXTERNAL	-	external procedure declaration
PROCEDURE	-	procedure declaration
FUNCTION	-	function declaration
INTRINSIC	-	built-in function

Executable statements

COND	-	all conditionals
100	-	all constants
:=	-	assignment
;	-	statement separation
BEGIN END	-	grouping statement
&	-	conditional variable separator

Figure 4. Constructs in the Relation Language

of complicated high level language constructs like Pascal's "with" statement and FORTRAN's "common" block. Variables defined as type *struct*, are treated as global variables, they may be defined anywhere in a program, but, they are accessed globally. Variables defined to be type *const* represent declared constants.

Statements

The four types of statements that can be identified in a relation program are: assignments, conditions, procedure calls and returns. Each of these statement types is discussed individually.

Assignments

Assignment statements are similar to those in any high level language, with one exception, any arithmetic operators appearing in the original source code are replaced by "&"'s. It is not necessary to know the identity of the original operator in order to perform structure metric analysis, so the substitution disguises the content of the original source.

Conditions

As with arithmetic operators, it is unnecessary to know the type of conditional represented in order to derive the flow of control that is present. One only needs to know that the condition existed, and the variables that are needed to evaluate the decision. Looping constructs are simply replaced by a condition statement and corresponding block of statements that the loop encompasses. Decision statements similar to Pascal's "case" and "if ... then (else)" are replaced by a condition statement, *cond*, that contains the original decision construct's boolean expression variables. A corresponding block of statements associated with the conditional includes those blocks of state-

ments from the original source code that are executed as a result of the conditional's value. This means that the statements from each of the specific choices in a Pascal *case* statement would be grouped together to form a single block of statements to be associated with a single condition statement.

Procedure Calls

Procedure calls are identical to those in other high level block structured languages like Pascal and 'C'.

Returns

The return statement in a relation language program is similar to one in the 'C' programming language and is used to represent the return value of a function.

Procedure Declarations

Procedures in a relation program must be declared prior to their use. Three methods of procedure declarations are available. The first method is to give the procedure heading, including a procedure identifier and a list of formal parameter names, followed by the relation language statements that are included in the procedure. This is the most common method of procedure declaration and is used to define procedures local to the current module. Declaring a procedure as external is the second method of procedure declaration. This method serves a dual purpose that allows relations to be defined over many separate modules and provides for the translation of subroutines that call each other. The final method provides for the declaration of intrinsic functions. The relation lan-

guage has no intrinsic functions and therefore requires all intrinsic functions called in the original source code to be defined in the corresponding relation program.

Observations

It is apparent that a relation program bears little resemblance to the source program from which is derived and that the relation language provides all of the necessary constructs to represent the code from a myriad of source languages. In this study the relation language is used to represent GPL source code in order to perform structure metric analysis. The translation from GPL to the Relation language is presented next.

The Translation From GPL to The Relation Language

Introduction

The Dialogue Management System provides a graphical editor for the creation and modification of system designs and the GPL relation language translator uses the same database as the editor for retrieval of SFD information. Much of the translation from GPL to relation language is straight forward and involves simple database information retrieval. The description of the translation is presented from the point of view of the target statements that need to be generated to correctly represent the flows of information and control for a single subroutine definition and declaration in the relation language. Since there are no global variables or constants in GPL, it is sufficient to discuss the translation of a single subroutine. The translation of the design, found in Appendix A, is located in Appendix B and the algorithm for translation follows:

```
Routine Translate_GPL_Design;
        for here each SFD;
          {
          generate procedure header;
          generate local variable declarations;
          Translate_SFD;
          }
    end;
```

Procedure Header Generation

Generation of the procedure header involves little processing other than information retrieval. The procedure name is retrieved from a database relation in addition to a list of the input and output parameters for the procedure. GPL parameters may be identified as input parameters, output parameters or as both input and output parameters. Since the relation language does not distinguish between input and output parameters, it is necessary to perform an intersection on the lists of input and output parameters and generate the resulting set of parameters for the procedure header.

Local Variable Declarations Generation

Generating local variable declarations involves the retrieval of the list of variables for the procedure and the generation of a declaration for each one.

Relation Language Code Generation for SFD Members

The term "member" refers to each symbol in a SFD. Many of the member types in a SFD require no code to be generated in the relation language. These member types include: start, return, databox, decision, data arc and control arc. Elimination of these member types leaves only those

member types for which there is a corresponding statement type in the relation language. The algorithm for translation of a SFD follows:

```
Routine Translate_SFD;
        for each function, DC-function, dialogue function
                and ICB;
          {
          generate_conditional;
          generate_code;
          }
    end;
```

One might believe that the decision member type ought not appear on this list due to the existence of the condition statement in the relation language, however, the decision symbol and the condition statement have different semantics. A decision symbol dictates the flow of control within an SFD, where a condition statement dictates the flow of information within a procedure. The condition statement contains the list of those variables whose values determine whether or not a certain statement gets executed. The statements that are effected by the values of "condition" variables are easily found during visual examination of a SFD. Program constructs like loops and "if ... then (... else)"s are quickly recognized. Control flow, however, is difficult to discern given the limited information available in the database, including which member each arc starts at, goes to and whether or not the arc has a conditional on it. Therefore, a backwards approach is taken to determine the flows of information. For each of the member types, function, DC-function, dialogue function and ICB, a single conditional is generated that contains each of the conditionals from any of the paths that might have been traversed to reach the current member. This generated conditional is formed by following all of the paths that end at the current member backwards to the start symbol. Each time an arc containing a conditional is traversed a decision node has been found and it must be determined whether or not its conditionals must be included in the generated conditional. All paths out of the decision node are followed until a return symbol or the original node is reached. The method for generating a conditional in algorithmic form is:

```
Routine generate_conditional;
        if ( not at the start symbol)
           {
           if ( current node is a decision node)
              {
              if ( conditionals_belong )
                 add conditionals;
              }
           follow all input arcs backwards;
           generate_conditional;
           }
     end;
```

If all of the paths out of the decision node lead to the original node, the conditionals are not included since the node will be executed regardless of the values of the condition variables, however, if any of the paths lead to a return symbol, the conditionals are included in the generated conditional. An algorithm for tracing conditionals is:

```
Routine conditionals_belong;
        for each outgoing arc;
           {
           follow paths forward;
           if (any path reaches a return symbol)
              include conditionals;
           else (path reaches node for which a conditional is being generated)
              do not include conditionals;
           }
     end;
```

Since there is only one start symbol per SFD and all paths eventually terminate at a return symbol, this algorithm is guaranteed to terminate. No check to see if a variable is already a part of the generated conditional is made because redundancy of flows of information is removed by the metric generation stage of the analyzer. The algorithm for code generation for DC-functions, functions, dialogue functions and ICBs follows:

```
Routine generate_code;
        if ( member is a DC-function or function)
          generate procedure call;
        if ( member is a dialogue function)
          generate appropriate updates to INPUT and OUTPUT structs;
        if ( member is an ICB)
          call target language translator;
    end;
```

Functions and DC-Functions

For the member types function and DC-function a condition statement and a procedure call are generated. The condition statement is generated as above, and the procedure call is generated by matching up the actual parameters with the corresponding formal parameters.

Dialogue Functions

For dialogue members it is necessary to declare two global structures INPUT and OUTPUT since, any input or output operation is actually an update to or an access to a global structure, standard input or standard output. For input dialogue operations an assignment statement of the following form is generated:

$$<variable> = INPUT$$

For output dialogue operations an assignment of the following form is generated:

$$OUTPUT = <variable>$$

Inner Code Blocks

For ICBs a conditional statement is generated, and a call is made to the translator for the target high-level language to output the proper relation language code for the code in the ICB. This call is possible since the contents of an ICB are syntactically and semantically correct code.

The Generation of Micro Metric Values

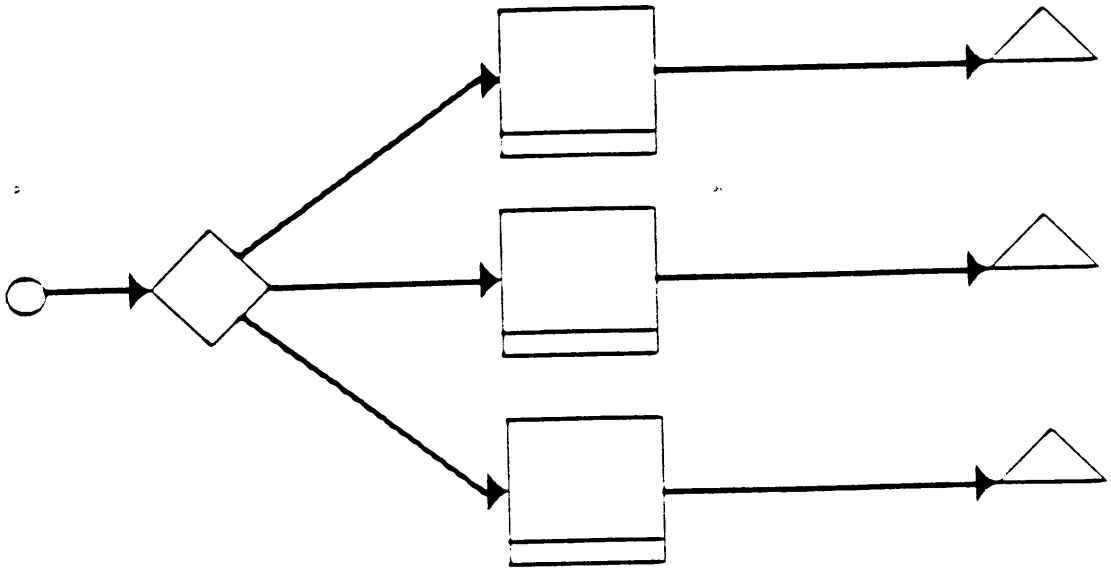
Micro metric value calculations are performed concurrently with the translation of GPL to relation language and are discussed in the following sections.

Length

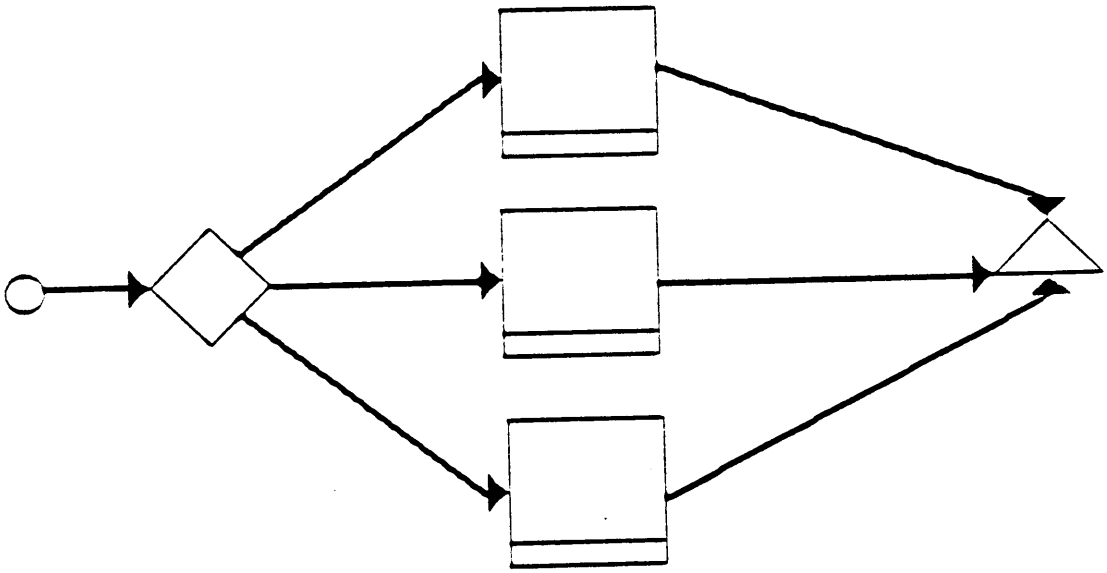
Length appears to be something difficult to measure in a graphical language and perhaps it is a meaningless measure, however, if one views the length of a design as a predictor of lines of code in the resulting system, a measure is feasible. Length of a GPL design is given by:

$$length = \#members + \text{length of ICB code} + \#decisions$$

The formula is derived from intuition. Each member of an SFD is likely to produce a line of code, regardless of the member type, so initially length is set to this value. Then, each time an ICB is encountered, the ICB length, returned by the target language's translator, is accumulated. Finally, the number of decisions is added to account for an additional label that is likely to be present in the resulting source code.



V(G) = 1



V(G) = 3

Figure 5. Multiple Exit Point Handling for Cyclomatic Complexity Calculation

McCabe's Cyclomatic Complexity

The cyclomatic complexity, $V(G)$, is perhaps the easiest micro metric value to calculate. Recall, from chapter two, that graph theory states that a graph's cyclomatic number is given by:

$$V(G) = E - N + 2$$

where,

$V(G)$	=	the cyclomatic number of a graph
E	=	the number of edges in a graph
N	=	the number of nodes or members in a graph

The value of $V(G)$ in GPL is initialized to the number of arcs minus the number of members. This alone, however, is not sufficient since GPL allows for any number of returns in a SFD. Cyclomatic complexity requires that the graph have unique entry and exit points. To account for this, each time a return symbol is encountered during translation, the cyclomatic number is increased by one, then instead of adding two at the end we only add one, to take out a single node for the unique exit point. Figure 5 on page 42 displays two SFDs, that are semantically equivalent, and their corresponding Cyclomatic Complexities, calculated without taking the multiple exit points into consideration. The complexities are different, even though the graphs are semantically the same.

Halstead's Software Science

Perhaps the most difficult micro metrics to calculate, due to the ambiguity in determining what is an operand and what is an operator, are Halstead's Software Science metrics. This study defines the required counts, unique and total operands and unique and total operators, on a graphical language. The counts of operands are straight forward and include anything that appears as text in a SFD: the supervisor name, parameter and variable names, variables that appear in conditionals,

variables from the databoxes and the names of called subroutines. The counts of operands are also modified by the high level language translator used to translate the code in the ICB's. The counts of operators in GPL are less intuitive than that of operands, however, bearing in mind Halstead's intuitive definition of the N value, the vocabulary of a routine, the counts become more obvious. The counts of operators in ICBs, performed by the high level language translator, are accumulated with those returned by the conditional parser, with identification of operators being performed as in conventional, textual languages. The number of members in a SFD and the number of control and data flow arcs in a SFD are also accumulated. Intuitively, the counts of members and arcs as operators is necessary since the members and arcs and their semantics are part of a designer's working vocabulary.

Conclusion

The methods presented do not represent the first attempt at translation and micro metric calculation, they are the result of many hours of thought and experimentation. A great deal of time was spent in attempting to make the translator behave logically in recognizing the familiar programming constructs in a SFD. The resulting solution is motivated by the recognition that GPL is not a block structured language and that it is very difficult to impose such a structure with the available information. Chapter four presents the results of metric analysis after applying the above transformation to GPL designs.

Chapter 4. Complexity Measurement

Introduction

In this chapter, the results of analyzing GPL design complexities are given. Complexity measures of GPL designs are presented along with equations, for each of the metrics, that allow the complexity of code corresponding to a GPL design to be predicted. An examination of GPL as a design tool is also briefly reported. A parallel measurement and prediction analysis of ADLIF, a general purpose PDL used in this study, is given as well.

The Experiment

The data in this study was collected from an assignment, given in a graduate level operating systems course consisting of twenty-two graduate students. Students simulated the management of consumeable and reusable operating system resources to detect and prevent deadlock, respectively. The banker's algorithm is used for deadlock prevention and knot detection algorithms are used for deadlock detection. In this study, one half of the class designed the program using GPL and the

other half used a general purpose textual design language, ADLIF, which is the subject of another complexity analysis study [SELC 87]. ADLIF is an attempt to structure pseudocode, i.e. a PDL. A copy of the assignment can be found in Appendix D. The assignment required the students to submit an initial design, one week prior to the assignment due date, a revised design, on the due date, and the Pascal source code and simulation results. Pascal was required to eliminate any differences resulting from using different programming languages. The revised designs were included as a part of the assignment for two reasons. First, to enable the evaluation of the two design tools in terms of the amount of change required to achieve a working system and second, to allow for the iterative refinement of the initial designs. Only the eighteen correct projects were used with nine of the projects coming from each group.

Data Preparation

In order to perform statistical the correlations of initial design to revised design and revised design to source, and to perform the regression analysis, it is necessary to have the same number of data observations, procedures, in the data being compared. It is possible and in fact likely that a design will not have the same number of routines as the source code. Often the source code will use many routines to perform the function of a single routine specified in the design. Another cause of extraneous routines in the source code may be the inability to refine a particular type of function in the design language like dialogue functions in GPL. Similarly, a routine may appear in a design, but, its function is combined into another routine in the resulting source code. It is necessary to incorporate the complexity measures of all of the routines in the source code into the data to be analyzed. When a routine exists in the source and does not have a corresponding routine in the design one accumulates the complexities of the more refined routines with the complexity of their parents. This is a valid operation since the design required the function to be performed and therefore its complexity is present in the design. The case where a routine is present in the design but not in the source code identifies a design that is not properly refined. One problem arises as a

result of the accumulation process. The complexity of the main program in the source code becomes unrealistic since many routines are accumulated into it. This occurs when programmers do not nest procedure declarations and as a result the only place for a routine's complexity to be accumulated is in the main program. Beyond programming style, it is possible that the language being used, for example GPL and 'C', may not allow procedure nesting and again the routine's complexity must be accumulated in the main program. As a result of language limitations and the programming style, the main program's complexity no longer reflects the actual complexity of the code. In this study the main programs were removed from the data prior to performing the statistical analysis. Three hundred and twenty-three procedures from eighteen projects were used in this analysis.

Data Presentation Information

Table 1 on page 48 contains the abbreviations that are used in the tables displayed throughout this study. An abbreviation for each of the nine metrics calculated is given.

Comparison of Initial and Revised Designs

Comparing the initial and revised designs provides a measure of the change required inorder to achieve a working system. This measure is meaningful only if there is a good correlation between revised design and source code [VINJ 87]. In this study there is a good correlation between revised design and source code, and that correlation is in the next section. Table 2 on page 49 displays the correlations of initial to revised designs for both the GPL and ADLIF groups. The correlations between the micro metrics in GPL designs are higher than those in ADLIF designs, however, the converse holds true for the structure and hybrid metrics. Neither of these results is significant due to the high degree of correlation found in all cases. One concludes that the design languages are

Table 1. Metric Abbreviations Used in Data Presentation

Metric	Abbreviation
Length	LOC
N	N
Volume	V
Effort	E
Cyclomatic Complexity	CC
Information Flow	INFO
Information Flow with Length	INFO-L
Information Flow with Effort	INFO-E
Information Flow with Cyclomatic	INFO-CC

Table 2. Correlations of Initial to Revised Designs

GPL								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.884	0.917	0.972	0.942	0.935	0.877	0.955	0.710	0.952

ADLIF								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.881	0.859	0.891	0.890	0.825	0.982	0.988	0.988	0.932

GPL Design Correlations with no Outliers								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.905	0.943	0.974	0.978	0.944	0.924	0.829	0.926	0.865

equally effective at helping the designer to create a good design, where a good design is one that accurately reflects the structure of the corresponding working source code. Looking at Table 6 on page 60, which gives the mean complexities of GPL and ADLIF designs, it is interesting to note that the complexity of GPL designs actually went down, for the structure and hybrid metrics, from the initial to the revised designs. Further investigation revealed that two procedure's complexities are responsible for the difference, removal of these routines produced complexities that are closer together. The two outlying procedures were found to be routines that had been reorganized and actually performed different functions in the revised design while keeping the same name as in the initial design. The revised mean complexities are found in Table 6 on page 60 and the revised correlations are in Table 2 on page 49.

Regression Analysis of GPL and ADLIF

Explanation

Regression analysis involves the comparison of sets of data between which there is some assumed inherent relationship. In simple linear regression one is concerned with a single independent and a single dependent variable and an attempt to derive a linear prediction, or regression, equation [WALR 78]. In this study, a simple linear regression is performed, with the complexity of design as the independent variable and complexity of the source as the dependent variable, in an attempt to derive equations which would allow a designer to estimate the complexity of the system being developed prior to its implementation. Regression analysis is performed for each of the calculated metrics.

GPL

Prior to the performance of the regression analysis on the GPL revised designs and GPL designed source code, a correlation between the two sets of values was performed to see if there was any relationship between the two. Table 3 on page 53 gives the results of the correlation. The high correlations indicate that a relationship between the data sets may exists, so the regression analysis is performed. Table 4 on page 54 contains the equations that result from the regression analysis of the GPL revised designs and the GPL designed source code. The column labeled *Coef* gives the value of the y-axis intercept and the slope of the regression line for the corresponding metric. The column labeled *Std Err* gives the standard error found in the calculation of the coefficient and the column labeled *t-Value* gives the value from the Student T distribution and is used for significance and confidence testing. The coefficient will fall within the range of plus or minus two times the standard error. A t-Value of greater than two generally represents ninety-five percent confidence that the corresponding coefficient is correct. The t-Values for each of the metric's slope are well above two and ninety-nine percent confidence in their values can easily be assumed. Ninety-nine percent confidence in all of the y-axis intercepts, except Cyclomatic and Information Flow with Cyclomatic, can also be assumed. The intercepts for the Cyclomatic complexity and the Information Flow Complexity combined with the Cyclomatic complexity can be assumed to be zero because of the low t-value [VINJ 87]. Figure 6 on page 55 gives a plot of the actual data observations, the regression line and the ninety-five percent confidence lines for the GPL information flow measure. The information flow metric was selected for this example and similar plots for each of the GPL measures are found in Appendix E. The prediction equation for a procedure's information flow complexity is as follows:

$$y = 1.103x + 205.167$$

where,

y = the predicted source code information flow complexity of the procedure

x = the calculated design information flow complexity of the procedure

With ninety-five percent confidence the predicted y value will fall within the confidence interval. Similar equations for each of the other metrics may be obtained by reading the coefficient values in Table 4 on page 54.

ADLIF

A correlation analysis between the ADLIF revised designs and the ADLIF designed source code was performed prior to doing the regression analysis to determine if there was any relationship between the two sets of data. Table 3 on page 53 contains the results of the correlation. The high correlations indicate that a relationship between the data sets may exist, so the regression analysis is performed. Table 5 on page 56 displays the equations that result from the regression analysis of the ADLIF revised designs and the ADLIF designed source code. The table is shown in the same format as for the GPL regression analysis and the values given represent the same items. The t-Values for each of the metric's slope are, once again, well above two and ninety-nine percent confidence in their values can is assumed. Ninety-five percent confidence in all of the y-axis intercepts is assumed and in many cases ninety-nine percent confidence can be assumed. Figure 7 on page 57 gives a plot of the actual data observations, the regression line and the ninety-five percent confidence lines for the ADLIF information flow complexity measure. Similar plots for each of the ADLIF measures are found in Appendix F. Note, when comparing the GPL and ADLIF plots that the scales may vary.

Conclusion

The results presented in this chapter indicate that it is possible, given the complexity of a GPL or ADLIF design, to predict the complexity of the corresponding source code. However, it is important to note that designs with different levels of refinement may produce different results. The

Table 3. Correlations of Revised Designs to Designed Source

GPL								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.78	0.702	0.66	0.508	0.793	0.808	0.788	0.737	0.752

ADLIF								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.849	0.834	0.843	0.749	0.711	0.901	0.887	0.832	0.8

Table 4. Regression Line Equations and Statistics for GPL Designs

GPL Regression Line Information				
		Coef	Std Err	t-Value
Length	Intercept Slope	3.878 0.826	0.871 0.055	4.454 15.105
N	Intercept Slope	22.258 1.141	8.781 0.096	2.535 11.947
Volume	Intercept Slope	211.697 1.639	42.367 0.154	4.997 10.643
Effort	Intercept Slope	12892.83 2.222	2666.116 0.311	4.836 7.15
Cyclomatic	Intercept Slope	-0.03 1.325	0.334 0.084	-0.09 15.799
Information Flow	Intercept Slope	205.167 1.103	77.814 0.066	2.637 16.629
Info. with Length	Intercept Slope	4985.403 1.278	1709.029 0.082	2.917 15.524
Info. with Effort	Intercept Slope	14976060 3.025	4711310 0.229	3.178 13.235
Info. with Cyclomatic	Intercept Slope	1284.167 1.539	735.014 0.111	1.747 13.812

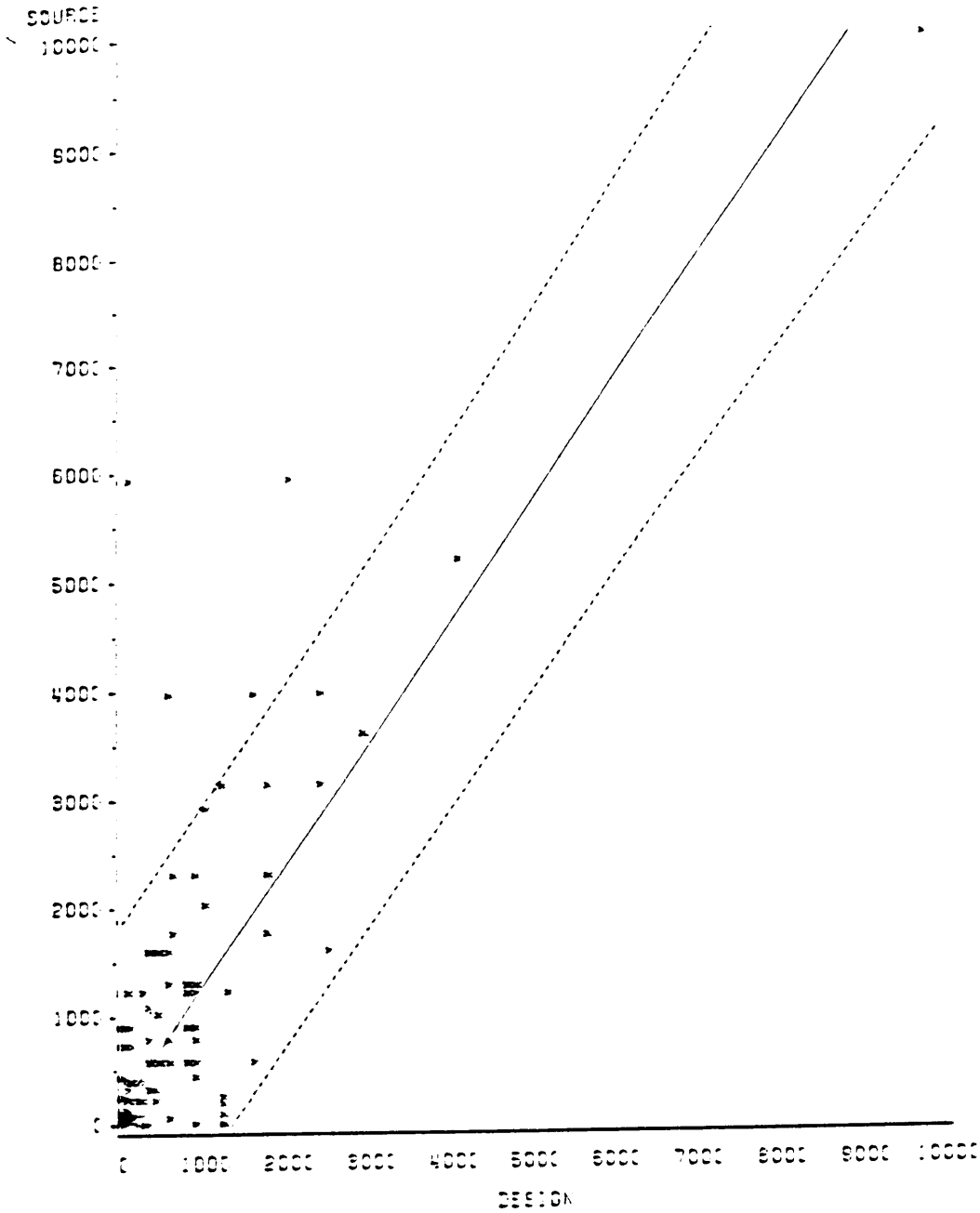


Figure 6. Regression Line and 95% Confidence lines for Information Flow Complexity in GPL

Table 5. Regression Line Equations and Statistics for ADLIF Designs

ADLIF Regression Line Information				
		Coef	Std Err	t-Value
Length	Intercept Slope	2.25 0.854	0.829 0.041	2.714 21.101
N	Intercept Slope	16.452 0.947	6.563 0.048	2.507 19.802
Volume	Intercept Slope	87.143 0.943	33.769 0.046	2.581 20.529
Effort	Intercept Slope	5698.626 1.079	2032.574 0.073	2.804 14.833
Cyclomatic	Intercept Slope	1.166 0.776	0.313 0.059	3.723 13.262
Information Flow	Intercept Slope	347.4512 0.707	106.084 0.026	3.275 27.197
Info. with Length	Intercept Slope	13294.48 0.48	2605.205 0.019	5.103 25.235
Info. with Effort	Intercept Slope	22514560 .712	5495677 0.036	4.097 19.633
Info. with Cyclomatic	Intercept Slope	3076.348 0.501	755.927 0.029	4.07 17.505

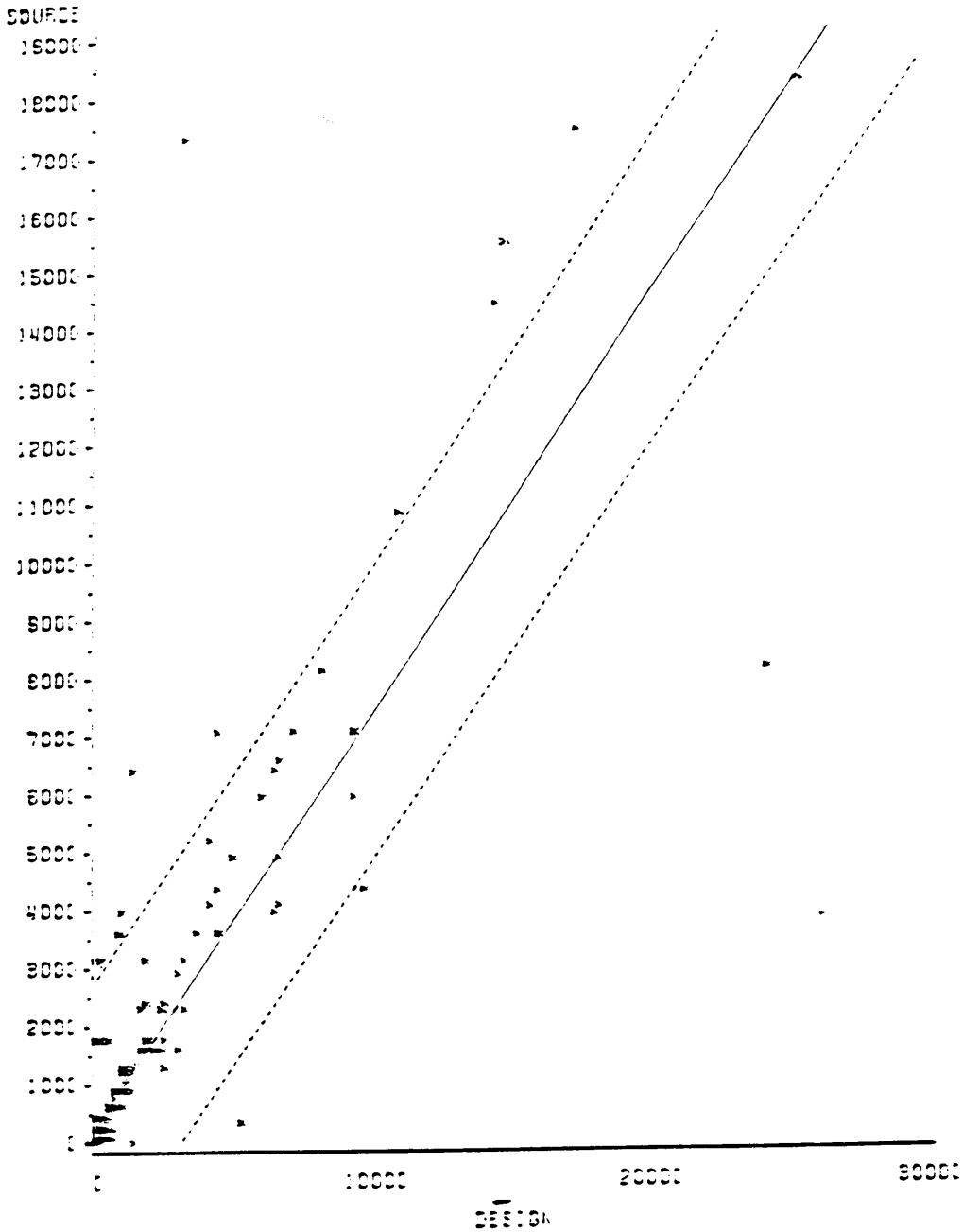


Figure 7. Regression Line and 95% Confidence lines for Information Flow Complexity in ADLIF

designs used in this study are at a very detailed level of refinement and the accuracy of the equations reflects the detail.

Chapter 5. Comparison of Design Tools

Introduction

In this chapter a comparison of GPL and ADLIF is presented. Specifically, the complexities of the designs and the complexities of the corresponding source code are compared. Both the initial and revised design complexities are given.

Comparison of GPL and ADLIF Design Complexities

Table 6 on page 60 presents the mean complexities of both the revised and initial designs for all of the GPL and ADLIF data. The complexities of the GPL designs are significantly lower than the ADLIF designs. The difference in complexities, however, is misleading due to the extreme difference in the nature of the languages. The semantics that can be expressed in a single symbol in GPL may take several keywords and symbols in ADLIF. Therefore, lower complexities of GPL designs are not necessarily an indication of less complex systems. Given a GPL and an ADLIF design for the same system, the GPL design should be measurably lower in complexity.

Table 6. Mean Complexities of Initial and Revised Designs

GPL									
Design	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
initial	14.032	85.948	251.656	6374.66	3.552	14129.62	1362948	808062000	337889
revised	14.519	87.89	261.617	6442.96	3.643	1003.13	28336.84	18113410	8164.23

ADLIF									
Design	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
initial	18.166	118.429	617.606	18489.24	4.2	16456.74	718024	678874000	48765.1
revised	19.536	128.219	675.426	21049.67	4.47	21447.1	882221	845066000	61844.7

GPL Mean Design Complexities with no Outliers									
Design	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
initial	13.375	80.678	233.974	5698.52	3.375	688.395	10431.23	7348219	2930.033
revised	14.151	84.618	246.842	6009.91	3.52	686.559	12050.82	8206303	3302.488

Comparison of GPL and ADLIF Designed Source Code

In comparing the GPL and ADLIF designed source code, the procedures were divided into five functional units, or modules, including: tape request processing, tape release processing, message sending processing, message receiving processing and deadlock detection. A sixth unit was defined as the complexity of the entire system. Table 7 on page 62 presents the average module complexities for both the GPL and ADLIF designed source code. The complexities of the GPL designed source code are considerably lower than those of the ADLIF designed source code. Table 8 on page 63 presents the average ratio of complexity of the GPL designed source to the complexity of ADLIF designed source. The ratios of the micro metrics are higher than those of the structure and hybrid metrics with an average ratio of 71.7%. The average ratio of the structure and hybrid metrics is 17.4%. These ratios and complexities indicate that GPL is the better tool for designing program structure. The structure and hybrid metric ratios in Table 8 on page 63 show that the structure of the GPL designed source code is nearly six times less complex than the ADLIF designed source code. Due to the small sample size it is necessary to perform a statistical significance test on the module complexities. Table 9 on page 64 gives the calculated t-values from the Student T distribution for each of the metrics in each module. In order to be eighty percent confident that the results are statistically significant the t-value must be greater than 1.337, for ninety percent confidence, 1.746 and for ninety-five percent confidence 2.12. In most of the cases the t-values ensure more than eighty percent confidence that the results are statistically significant. The *send* and *deadlock* module's t-values are lower than the eighty percent confidence level in many cases and therefore those results are not considered significant. Concentrating only on the entire system's confidence measures, the **all** module, there is better than ninety percent confidence, except for the effort and cyclomatic complexities, that the results are statistically significant.

Table 7. Mean Module Complexities for GPL and Adlif Designed Pascal Source Code

GPL

Module	LOC	N	V	E	CC	INFO
all	332.778	2438.222	12553.44	515028.5	95.000	122431400
request	100.889	749.556	3781.000	147167.3	33.778	1153413
release	128.000	963.556	4839.444	186975.2	40.778	2599250
send	52.444	391.667	1963.222	81965.9	14.222	46490.44
receive	66.667	537.667	2686.556	113073.8	22.222	1269498
deadlock	65.333	519.556	2621.111	124036.3	19.222	165059.7

Module	INFO-L	INFO-E	INFO-CC
all	4.923059e10	6.570745e13	1.277843e10
request	1.931443e8	2.037432e11	5.611805e7
release	4.329576e8	4.507008e11	1.253695e8
send	1.210397e7	2.269016e10	3.382792e6
receive	2.165298e8	2.458338e11	6.926457e7
deadlock	2.472898e6	3.753299e9	6.554804e5

ADLIF

Module	LOC	N	V	E	CC	INFO
all	412.778	2981.222	15588.11	612871.9	106.00	319512544
request	140.667	1052.222	5388.889	199651.3	45.333	3035089
release	184.889	1355.666	6977.556	247801.3	54.000	17837226
send	99.222	749.556	3799.333	167877.7	26.222	7370219
receive	100.333	740.556	3712.000	49763.6	27.667	6031054
deadlock	91.778	691.888	3536.778	133350.0	26.000	5641381

Module	INFO-L	INFO-E	INFO-CC
all	1.488619e11	2.037372e14	3.820597e10
request	4.787583e8	6.034352e11	1.574458e8
release	4.065810e9	5.220383e12	1.217569e9
send	1.674139e9	1.864659e12	3.945642e8
receive	1.097365e9	1.268334e12	2.797951e8
deadlock	1.206509e9	1.238027e12	3.358399e8

Table 8. Average Ratio of Complexities of GPL to ADLIF Designed Source

LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
0.687	0.707	0.696	0.744	0.75	0.194	0.172	0.16	0.169

Table 9. t-values from the Student T distribution

Module	LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
all	1.982	2.165	2.293	1.408	1.359	1.880	1.885	1.895	1.868
request	2.226	2.574	2.791	2.073	2.220	1.736	1.568	1.811	1.734
release	2.660	2.739	2.990	2.003	2.169	2.028	1.960	1.773	1.852
send	1.638	1.670	1.745	1.638	1.453	1.280	1.183	1.259	1.213
receive	2.819	2.877	2.938	2.943	2.691	1.709	1.497	1.655	1.576
deadlock	0.762	0.640	0.697	0.413	0.355	1.279	1.321	1.318	1.257

Conclusion

In this chapter results have been presented that indicate that systems designed in GPL are significantly less complex than systems designed in ADLIF thereby adding quantitative support to the work done by cognitive science researchers which finds that the human brain thinks and works more effectively in images. The inability to compare GPL and ADLIF designs directly is also shown.

Chapter 6. Additional Measurements

Introduction

In this chapter the intermetric correlations for the GPL and ADLIF revised designs and the GPL and ADLIF designed source code are displayed to replicate the results of two earlier studies [HENS 81] [KAFD 84]. Also presented is a preliminary data set that was collected and found to be inadequate for analysis. An explanation of why the data was inadequate and how the inadequacies came about is given. Also shown is a comparison of the source code written for the first data set.

Intermetric Correlations

Table 10 on page 68, Table 11 on page 69, Table 12 on page 70 and Table 13 on page 71, present the intermetric correlations for GPL and ADLIF revised designs and GPL and ADLIF designed source code. Both sets of GPL correlations and the ADLIF revised design correlations reflect the results of another study with the micro metrics having high correlations to the other micro metrics and low correlations with the structure and hybrid metrics and the structure and hybrid metrics

having high correlations with the other structure and hybrid metrics and low correlations with the micro metrics [HENS 81]. These correlations indicate that the structure and hybrid metrics measure different properties of software than the micro metrics. The fourth set of correlations, ADLIF designed source code, demonstrates the same relationship between micro metrics and other micro metrics and structure and hybrid metrics with other structure and hybrid metrics, however, the correlations between the micro metrics and the structure and hybrid metrics are not as low as in previous studies. It is unclear why the correlations are this way.

Table 10. Intermetric Correlations for GPL Revised Designs

GPL Revised Designs									
Metric	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
LOC									
N	0.855								
V	0.826	0.891							
E	0.636	0.684	0.702						
CC	0.902	0.830	0.790	0.603					
INFO	0.033	0.359	0.224	0.225	0.096				
INFO-L	0.245	0.547	0.405	0.376	0.315	0.914			
INFO-E	0.179	0.480	0.341	0.437	0.278	0.815	0.922		
INFO-CC	0.205	0.518	0.378	0.366	0.312	0.899	0.983	0.957	

Table 11. Intermetric Correlations for GPL Designed Pascal Source Code

GPL Designed Source									
Metric	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
LOC									
N	0.897								
V	0.897	0.997							
E	0.814	0.941	0.943						
CC	0.700	0.763	0.767	0.763					
INFO	0.223	0.221	0.227	0.203	0.176				
INFO-L	0.450	0.427	0.440	0.435	0.372	0.919			
INFO-E	0.485	0.529	0.548	0.613	0.496	0.735	0.901		
INFO-CC	0.351	0.373	0.382	0.405	0.478	0.819	0.888	0.883	

Table 12. Intermetric Correlations for ADLIF Revised Designs

ADLIF Revised Designs									
Metric	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
LOC									
N	0.961								
V	0.965	0.995							
E	0.825	0.905	0.889						
CC	0.432	0.538	0.500	0.654					
INFO	0.258	0.236	0.252	0.120	-0.080				
INFO-L	0.439	0.397	0.423	0.248	-0.038	0.889			
INFO-E	0.521	0.474	0.508	0.338	0.006	0.777	0.964		
INFO-CC	0.486	0.453	0.477	0.317	0.040	0.695	0.900	0.963	

Table 13. Intermetric Correlations for ADLIF Designed Pascal Source Code

ADLIF Designed Source									
Metric	**LOC**	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
LOC									
N	0.960								
V	0.956	0.992							
E	0.780	0.878	0.904						
CC	0.829	0.882	0.843	0.702					
INFO	0.703	0.699	0.620	0.405	0.746				
INFO-L	0.685	0.682	0.600	0.388	0.735	0.998			
INFO-E	0.692	0.689	0.608	0.395	0.740	0.999	1.000		
INFO-CC	0.682	0.680	0.597	0.385	0.734	0.997	1.000	0.999	

Preliminary Data Set

Introduction

A preliminary data set was collected prior to the collection of the data set used in this study. The set of data came from a senior level software engineering class whose curriculum consisted of design methodologies and software development and analysis. The class was divided into two sections, as in the data set analyzed above, and designed their assignments using the two design languages. The GPL group used a partially completed automated version of GPL for a design tool. Several key parts of the automated version of GPL were not working, including SFDs, dialogue functions and computational functions. As a result no low level design could be performed and the resulting design data had no relation to the corresponding source code. In the next few sections an analysis of the available information is given.

The Experiment

The class of fourteen seniors was asked to implement the "robots" game that is available on some UNIX systems. They were required to submit a design of their system one week prior to the due date of the entire system. A copy of the assignment is located in Appendix G.

Data Analysis

Table 14 on page 74 shows the correlations of design to source code for GPL and ADLIF. While the correlations for the ADLIF group are similar to those in the previous data set, indicating a strong relationship between design and source, the GPL correlations, by virtue of their low values, indicate no relationship and therefore, no regression analysis was performed[VINJ 87]. However,

a comparison of the GPL and ADLIF designed source code is presented. As in the previous data set, the source code was partitioned into functional units, or modules. The complexities of the modules were averaged and the results are given in Table 15 on page 75. As in the previous data set, the complexities of the GPL designed source code are significantly lower than the complexities of the ADLIF designed source code. Table 16 on page 76 shows the ratio of the average complexity of the GPL designed source code modules to the average complexity of the ADLIF designed source code modules and as before their is a measurable difference between the ratios of the micro metric complexities and the structure and hybrid metric complexities. The mean ratio for the micro metric complexities is 76.5%, which is within 5% of the ratios found in the previous data set, and the ratio for the structure and hybrid metrics is 4.4%, which is much less than that of the previous data set but none the less repeats the trend for measurably lower structure and hybrid metric complexities in GPL designed source code. Table 17 on page 77 presents the calculated t-values for the modules and the results are very close to those from the previous data set. However, one must down-play the significance of these results since there was such a low correlation between GPL design and GPL designed source code and as a result of the low correlation it is difficult to say how much of the difference is due to the language used for design as opposed to programmer ability.

Conclusion

This chapter shows that the complexity measures of GPL designs replicate the results of another study with the micro metrics correlating well with the other micro metrics and not the structure and hybrid metrics. The additional data set demonstrates that some minimum level of refinement of design is necessary in order to perform complexity analysis.

Table 14. Correlations of Designs to Designed Source Code

GPL								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.403	.344	0.311	0.273	0.422	0.154	0.268	0.802	0.55

ADLIF								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.774	0.774	0.832	0.689	0.627	0.657	0.588	0.793	0.832

Table 15. Mean Module Complexities for GPL and Adlif Designed Pascal Source Code

GPL

Module	LOC	N	V	E	CC	INFO
all	337.375	2669.50	13580.62	759748.8	115.375	4.317741e8
module1	66.625	482.25	2390.750	21498.1	17.250	1.233739e5
module2	73.250	750.00	4011.125	341337.9	34.375	7.588529e5

Module	INFO-L	INFO-E	INFO-CC
all	1.596113e11	3.516355e14	5.428607e10
module1	9.904538e6	2.436942e10	2.927187e6
module2	7.500602e7	4.115091e11	3.320601e7

ADLIF

Module	LOC	N	V	E	CC	INFO
all	475.333	3374.167	17006.50	835998	125.167	3.221515e9
module1	124.500	782.667	3714.500	166930	20.500	6.772004e6
module2	127.167	1094	5501.834	276823	45.333	1.084630e8

Module	INFO-L	INFO-E	INFO-CC
all	1.719255e12	3.076734e15	4.879234e11
module1	1.028491e9	1.946416e12	1.704274e8
module2	2.696170e10	5.111443e13	9.861835e9

Table 16. Average Ratio of Complexities of GPL to ADLIF Designed Source

LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
0.607	0.698	0.724	0.957	0.841	0.053	0.035	0.045	0.044

Table 17. t-values from the Student T distribution

Module	LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
all	3.556	2.469	2.492	0.534	0.720	2.274	2.257	2.386	2.030
module1	2.575	2.229	1.893	0.644	0.902	1.826	1.819	1.498	1.694
module2	2.308	1.750	1.489	0.67	1.201	1.317	1.289	1.292	1.286

Chapter 7. Conclusions

Conclusions

This study was begun with a twofold purpose. First, to determine whether the complexity of a graphical language could be measured, using established software quality metrics and second, to explore the hypothesis that systems designed using a graphical language are less complex than systems designed using a textual design language.

In Chapter three a technique for measuring GPL design complexities was presented. The technique involves the translation of a GPL design into an intermediate language, the relation language, from which an existing analyzer draws the needed information to produce the measures. Then in chapter four the results of measuring GPL designs are given. Chapter four also gave a set of equations to predict, with more than 95% confidence, the complexity of source code. Plots of the equations and the confidence intervals are contained in the appendix. These equations will allow the selection of the single least complex design, from a group of designs, that perform the same task and it will serve to shorten the design-code-measure- redesign cycle to a design-measure-redesign cycle.

Chapter five compared GPL designed source code with ADLIF designed source code. The mean complexities of GPL designed source code were significantly lower than the mean complexities of ADLIF designed source code and a statistical significance test revealed more than ninety percent confidence in the results for most metrics. Taking a closer look showed that the GPL designed source code has far less complex structure, indicating its superiority over ADLIF for designing system structure. These results support the hypotheses of cognitive science researchers who state that humans think in images and that the human brain works more effectively in images.

In summary, with a very limited dataset, we can:

- Measure GPL Design Complexities
- Predict the complexity of GPL designed source code with 95% confidence
- Show less complex source code
  - micro metrics are 71.7% as complex in GPL designed source
  - structure and hybrid metrics are 17.4% as complex in GPL designed source
  - 95% confidence that the results are statistically significant

These results indicate that GPL is a much better tool than ADLIF for designing system structure.

The design languages were presented to the students in the form of written documents. Soon after the documents were handed out a help session was held to answer any questions concerning their use. Many questions were asked in regards to ADLIF and very few were asked in regards to GPL. This leads to one final conclusion that GPL is easier to learn to use than ADLIF.

Future Work

There is much work to be done in the measurement and comparison of graphical and textual design languages. First, it is necessary to collect and analyze more data. The initial findings indicate that GPL is a better design tool than ADLIF in terms of the complexity of designed source code, but a much larger set of data is necessary. Second, GPL needs to be tested against other textual design languages before it can be stated that graphical design languages are better than textual design languages. The complexities of other graphical design languages need to be measured and compared to both textual design languages and to GPL. Comparison of graphical design languages will help to identify the most effective features of graphical design languages and could lead to the definition of a standard graphical design language. Finally, more historical data relating to the design process needs to be collected including: design time, implementation time and debugging time.

Bibliography

Belady, L., Lehman, M., "A Model of Large Program Development", *IBM Systems Journal*, No. 3, 1976, 225-252

Benzon, B., "The Visual Mind and The MacIntosh", *Byte*, January 1985

Bohl, M., *Flowcharting Techniques*, Palo Alto, CA, Science Research Associates, 1971

Brooks, F. JR., *The Mythical Man Month*, Reading, MA, Addison-Wesley Publishing Company, 1975

Brown, G., Herot, C., Souza, P., "Program Visualization: Graphical Support for Software Development", *IEEE Computer*, August 1985

Callan, J.E., *Behavioral Demonstrations: An Approach to Rapid Prototyping and Requirements Execution.*, Masters Thesis, Virginia Tech, Department of Computer Science, 1985

Conte, S.D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Models*, Menlo Park, CA, The Benjamin/Cummings Publishing Company, Inc., 1986

Fairley, R., *Software Engineering Concepts*, New York, Mcgraw-Hill Book Company, 1985

Glinert, E., Tanimoto, S., "Pict: An Interactive Graphical Programming Environment", *IEEE Computer*, November 1984

Halstead, M., *Elements of Software Science*, New York, NY, Elsevier North Holland, Inc., 1977

Hartson, H.R., *Advances in Human-Computer Interaction*, Norwood, NJ, Ablex Publishing Company, 1985

Henry, S., *Information Flow Metrics for the Evaluation of Operating Systems' Structure*, Ph. D. Disertation, Iowa State University, Computer Science Department, 1979

Henry, S.M., Kafura, G.D., "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, September 1981

Henry, S.M., Kafura, G.D., Harris, K., "On the Relationships Among Three Software Metrics", *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981

Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information fo Researchers", *Journal of Systems and Software*, 1988 (to appear)

Kafura, D., Canning, J., "The Independence of Software Metrics Taken at Different Life-Cycle Stages", *Proceedings: Ninth Annual Software Engineering Workshop, NASA Goddard Space Flight Center*, November 1984

Kafura, D., Henry, S., "Software Quality Metrics Based on Interconnectivity", *Journal of Systems and Software*, Vol. 2, 1982

McCabe, T., "A Complexity Measure", *IEEE Transactions on Software Engineering*, December 1976

McClure, C., "A Model for Program Complexity Analysis", *Proceedings Third International Conference on Software Engineering*, Atlanta, GA, May 1978, 149-157

Mills, H., "Mathematical Foundations for Structured Programming", Federal System Division, IBM Corp., Gaithersburg, MD, FSC 72-6012, 1972

Moriconi, M., Hare, D., "Visualizing Program Designs Through Pegasys", *IEEE Computer*, August 1985

Ramsey, H. Rudy, Atwood, Michael E., Van Doren, James R., "Flowcharts Versus Program Design Languages: An Experimental Comparison", *Communications of the ACM*, June 1983

Selig, C.L., *ADLIF - A Structured Design Language for Metric Analysis*, Masters Thesis, Virginia Tech, Department of Computer Science, June 1987

Smith, E.C., *System Support for Design and Development Environments*, Masters Thesis, Virginia Tech, Department of Computer Science, 1986

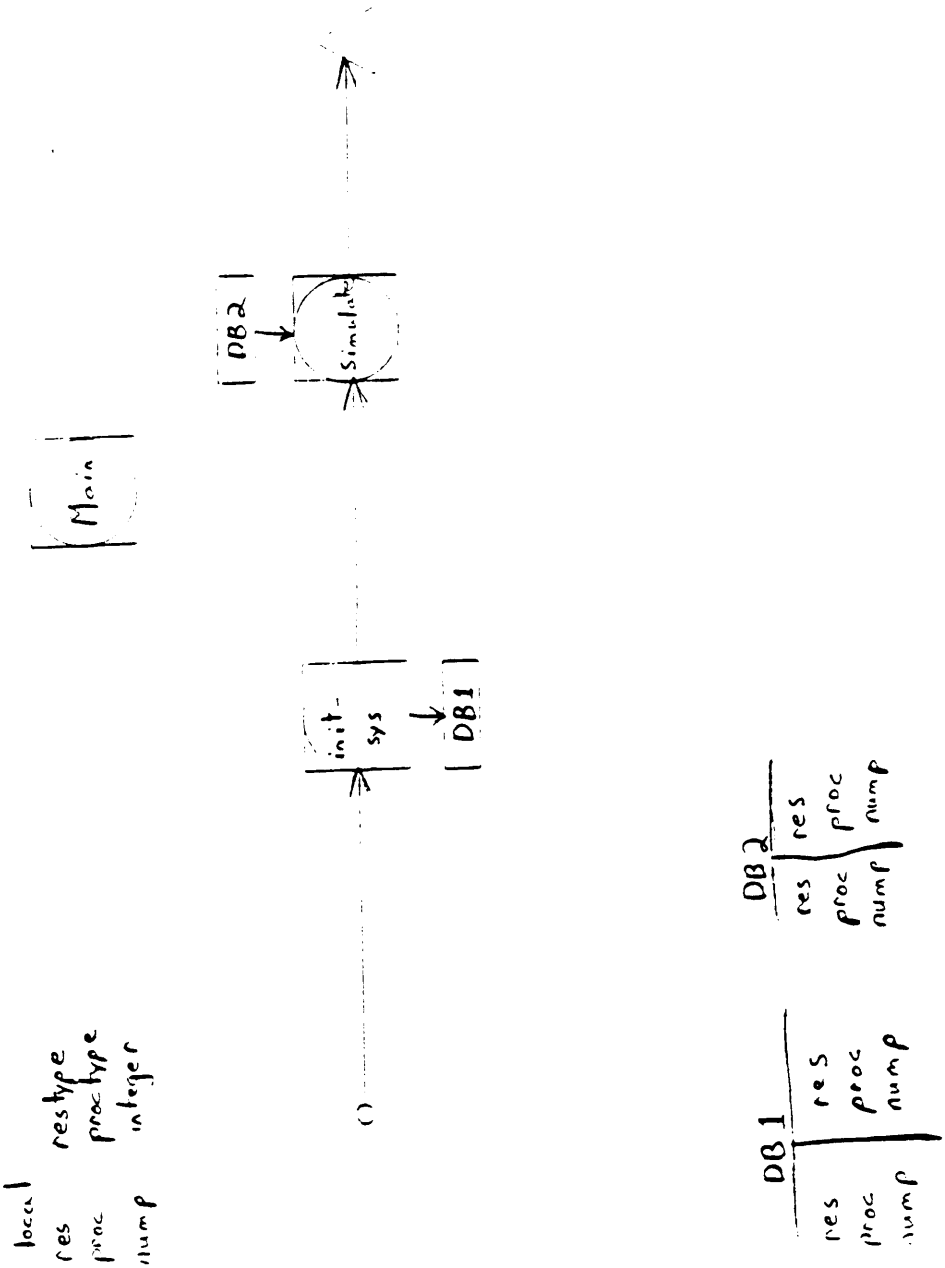
Vining J., Consultant, *Virginia Tech Statistics Laboratory*, personal discussion, 1987

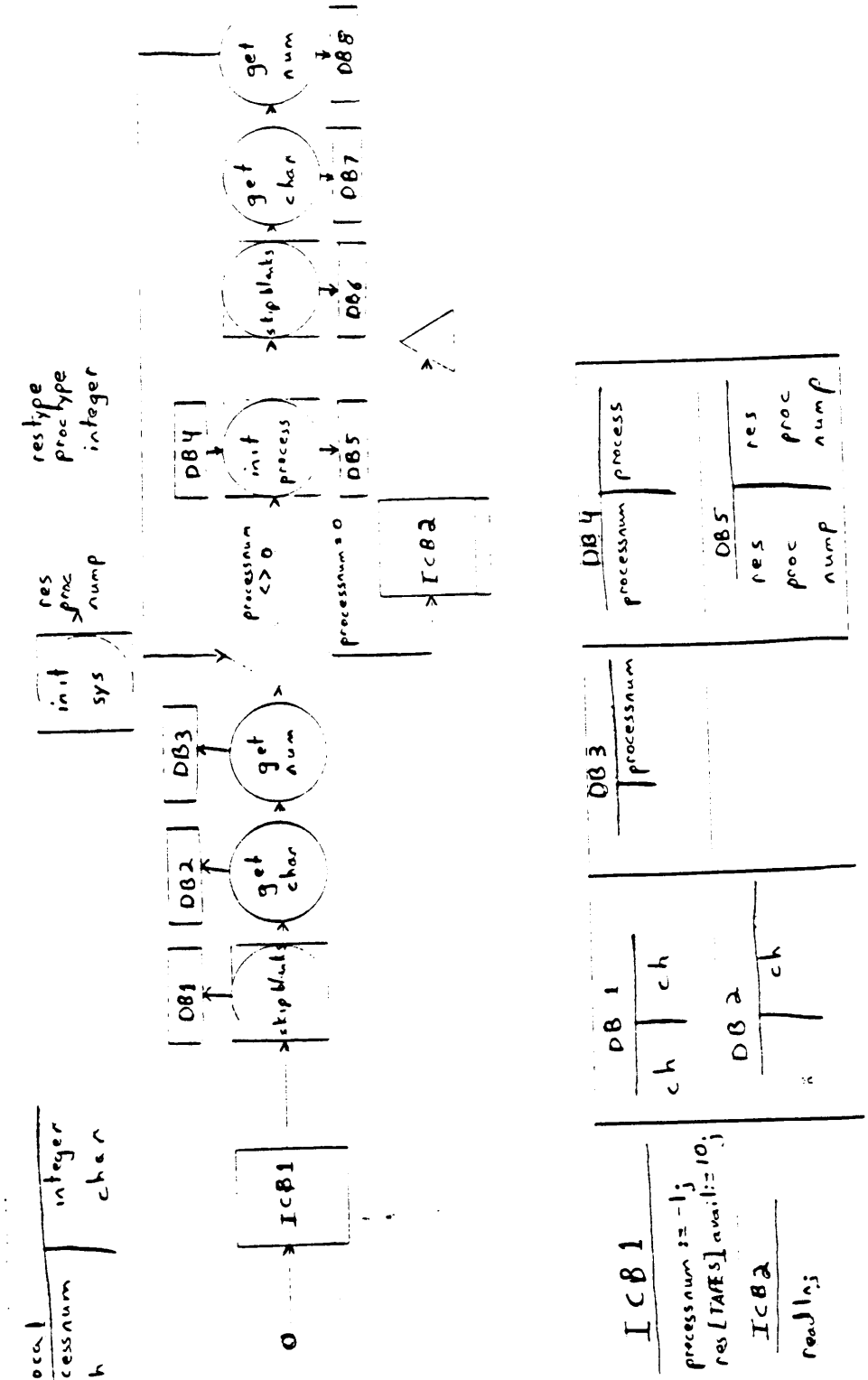
Walpole, Ronald E., Myers, Raymond H., *Probability and Statistics for Engineers and Scientists*, New York, NY, Macmillan Publishing Co., Inc., 1978

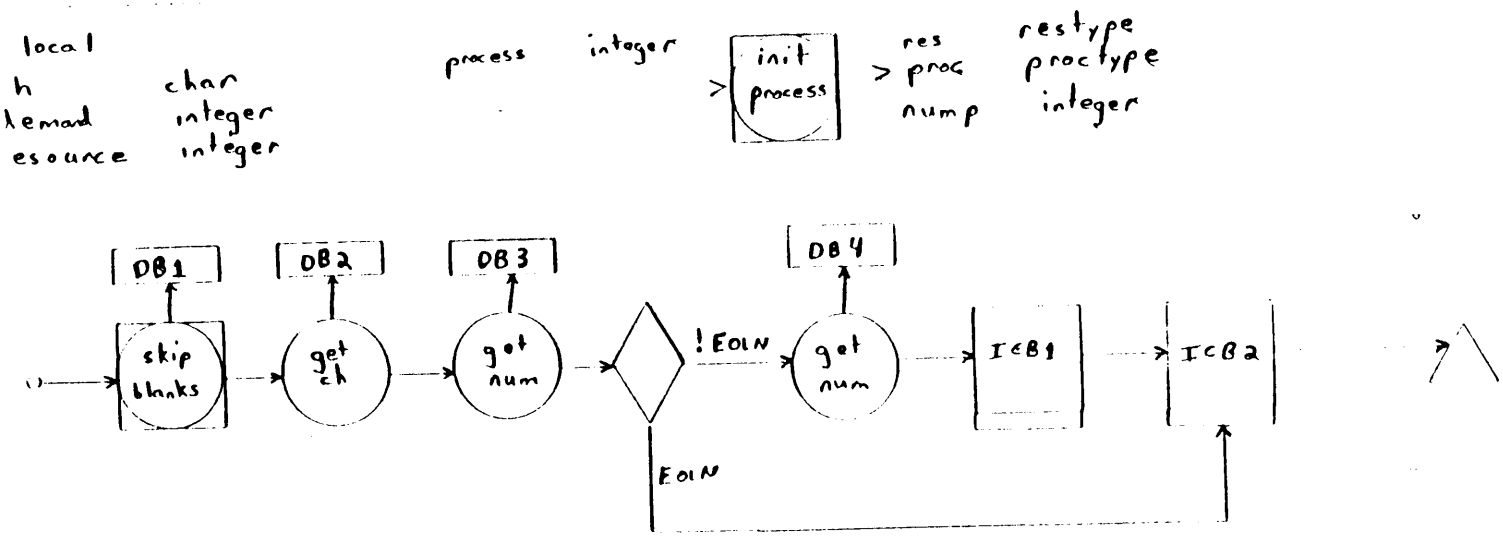
Woodfield, S., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph. D. Disertation, Purdue University, Computer Science Department, 1980

Yau, Stephen S., Tsai, Jeffery J.-P., "A Survey of Software Design Techniques", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 6, June 1986

Appendix A. Sample GPL Design


local
res restype
proc proctype
nump integer
Main
init-sys
DB1
DB2
Simulate
DB1
res res
proc proc
nump nump
DB2
res res
proc proc
nump nump


local
processnum
ch
integer
char
0
ICB1
DB1
DB2
DB3
strip blanks
get char
get num
init sys
res
proc
nump
restype
proctype
integer
processnum <> 0
processnum = 0
DB4
init process
DB5
ICB2
strip blanks
get char
get num
DB6
DB7
DB8
ICB1
processnum := -1;
res[TAPES].avail := 10;
ICB2
read(ch);
DB 1
ch
ch
DB 2
ch
DB 3
processnum
DB 4
processnum
process
DB5
res
proc
nump
res
proc
nump



ICB 1

```
new (res[resource].toarc]);
res[resource].toarc ^.pnum := process;
res[resource].toarc ^.next := nil;
proc[process].resources[resource].producer := true;
proc[process].resources[resource].requested := false;
proc[process].resources[resource].allocated := false;
```

ICB2

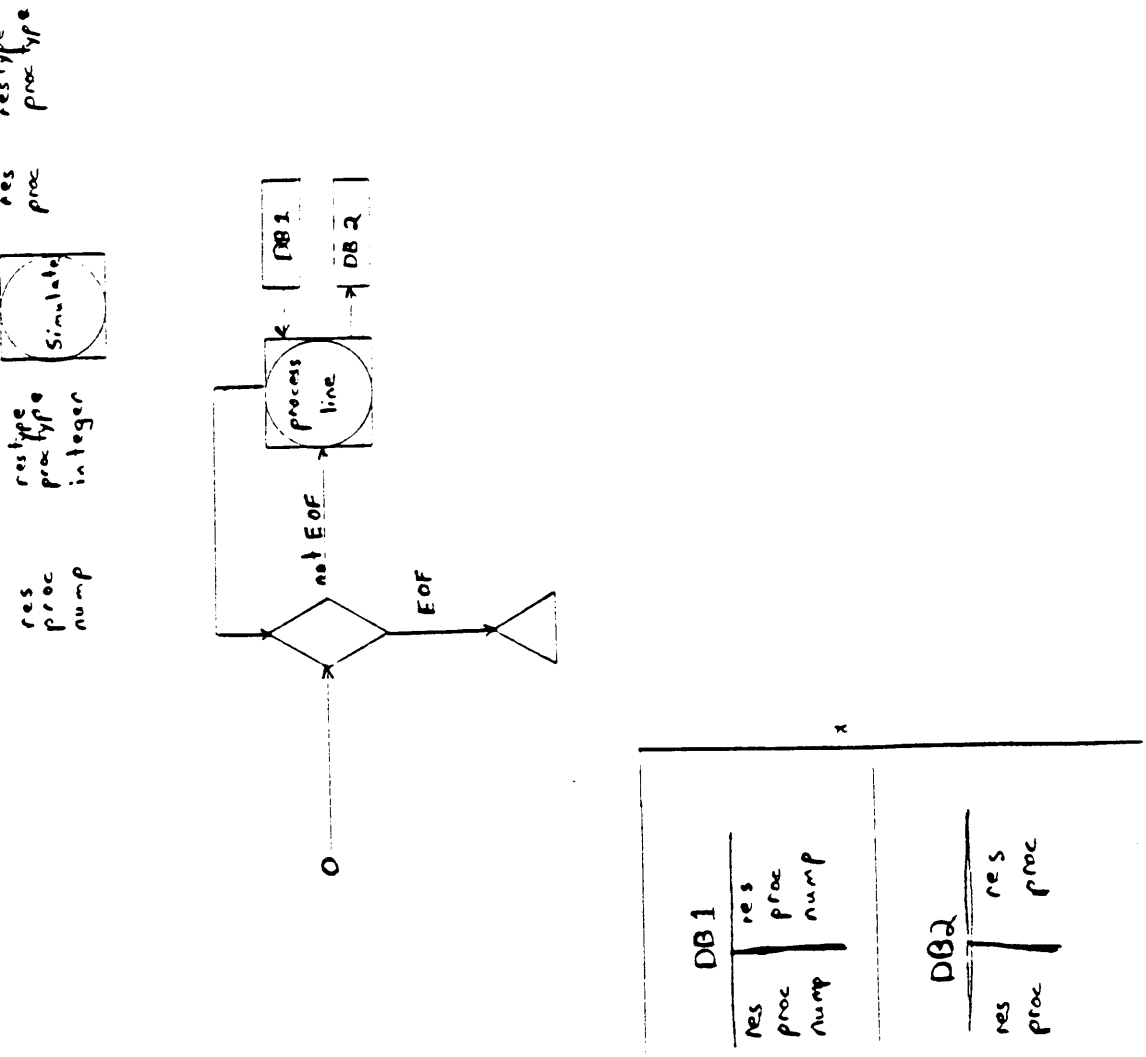
```
proc[process].demand := demand;
proc[process].alloc := 0;
nump := nump+1;
proc[process].req := 0;
```

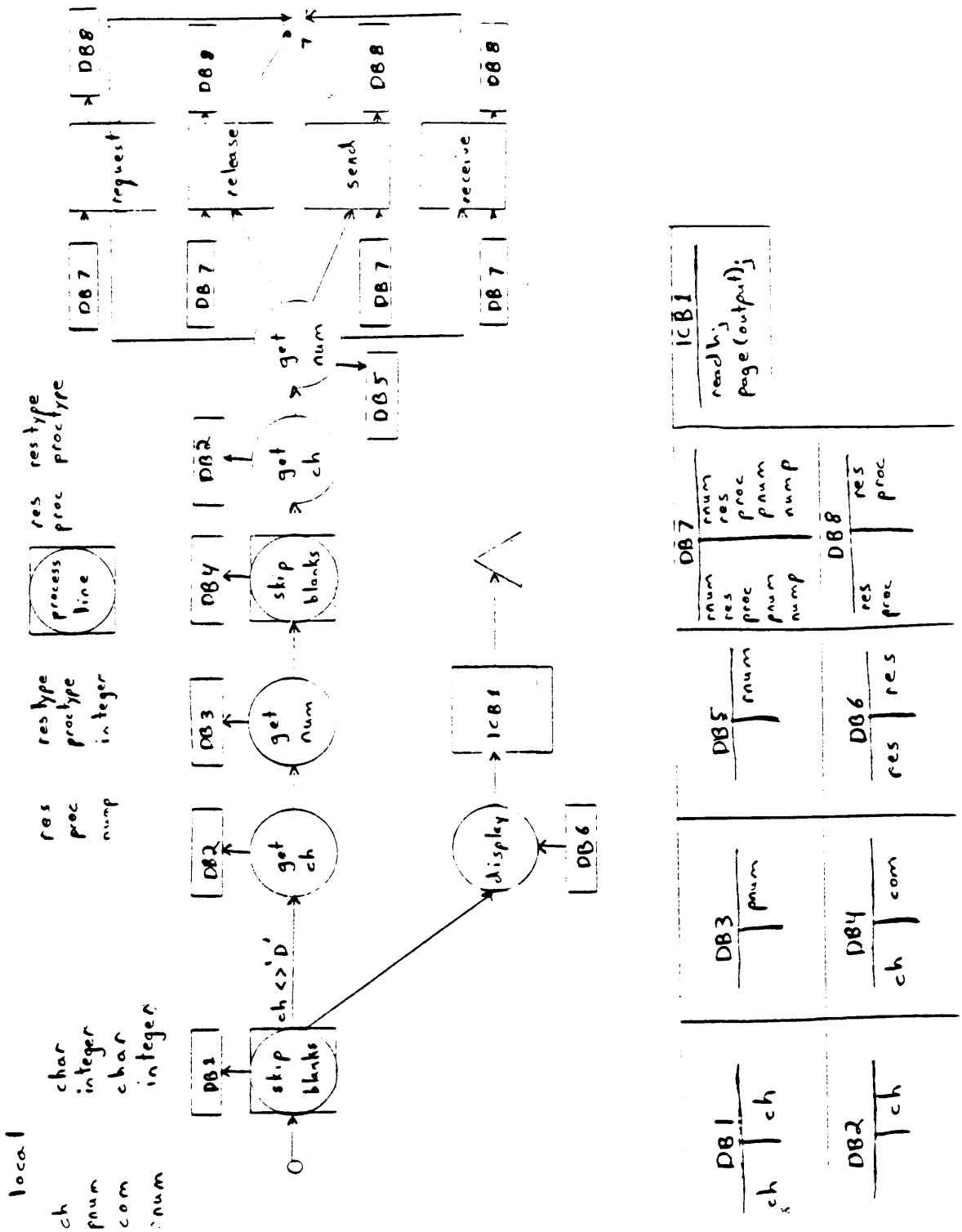
DB1	
ch	ch

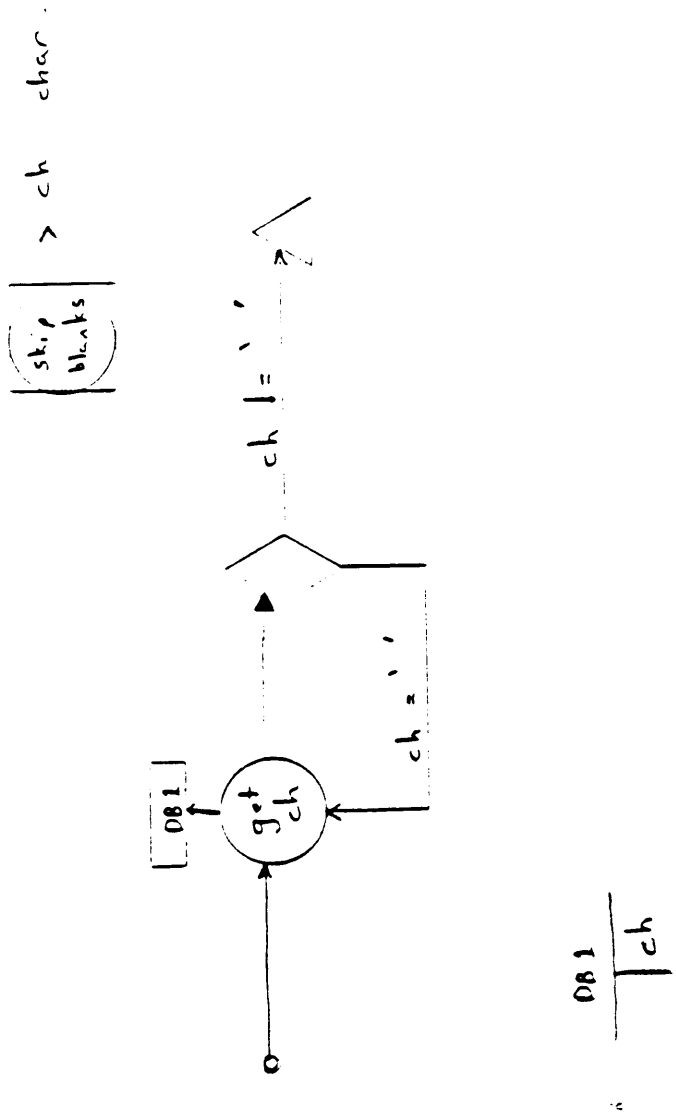
DB2
ch

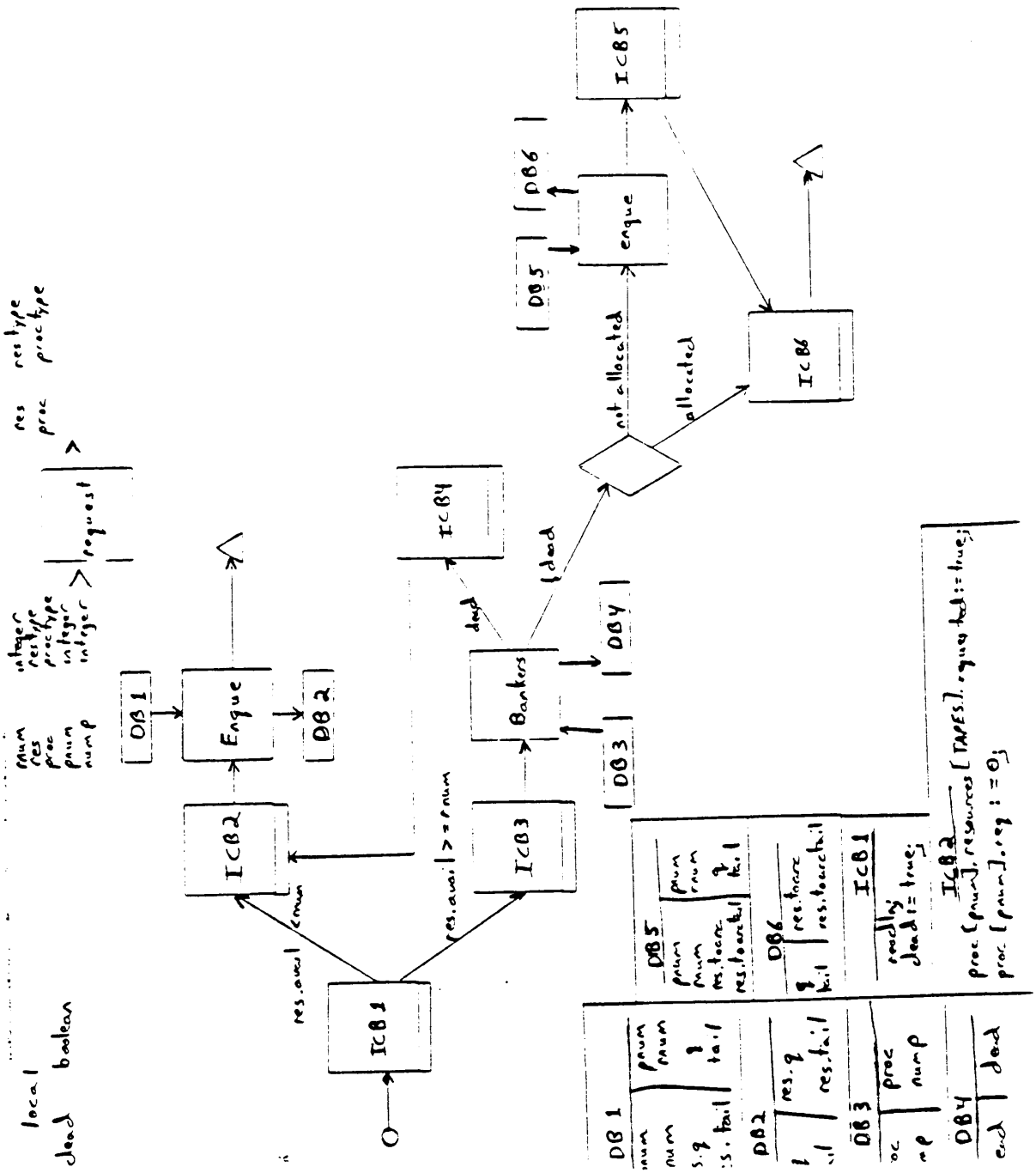
DB3
demand

DB4
resource


res
proc
num
res type
proc type
integer
Simulate
res
proc
res type
proc type
not EOF
process
line
DB 1
DB 2
EOF
DB 1
res
proc
num
res
proc
num
DB2
res
proc
res
proc


local
ch
pnum
com
num
char
integer
char
integer
res
proc
nump
res type
proc type
integer
res
proc
res type
proc type
DB1
DB2
DB3
DB4
DB5
DB6
DB7
DB8
skip blanks
ch <> 'D'
get ch
get num
skip blanks
process line
get ch
get num
display
ICB1
request
release
send
receive
ICB1
readln;
page(output);
DB1
ch
ch
DB2
ch
DB3
pnum
DB4
ch
com
DB5
num
DB6
res
res
DB7
num
res
proc
pnum
nump
DB8
res
proc


skip blanks
> ch
char
DB 1
get ch
ch = ' '
ch ≠ ' '
DB 1
ch


local
dead boolean
num
res
proc
pnum
numP
integer
restype
proctype
integer
integer
request
res restype
proc proctype
DB 1
Enque
DB 2
ICB2
res.avail
emun
ICB1
res.avail >= pnum
ICB3
Bankers
dead
ICB4
DB3
DB4
dead
not allocated
allocated
enque
DB5
DB6
ICB5
ICB6
DB 1
pnum
num
num
s.q
s.tail
DB2
res.q
res.tail
DB3
proc
numP
DB4
dead
DB5
pnum
num
res.tocc
res.tocctail
pnum
num
q
tail
DB6
res.tocc
res.tocctail
q
tail
ICB1
readly;
dead := true;
ICB2
proc (pnum], resources [TAPES]. requ dead := true;
proc [pnum].req := 0;

ICB3

```
proc[pnum].alloc := proc[pnum].alloc + rnum;
proc[pnum].resources[TAPES].requested := false;
```

ICB4

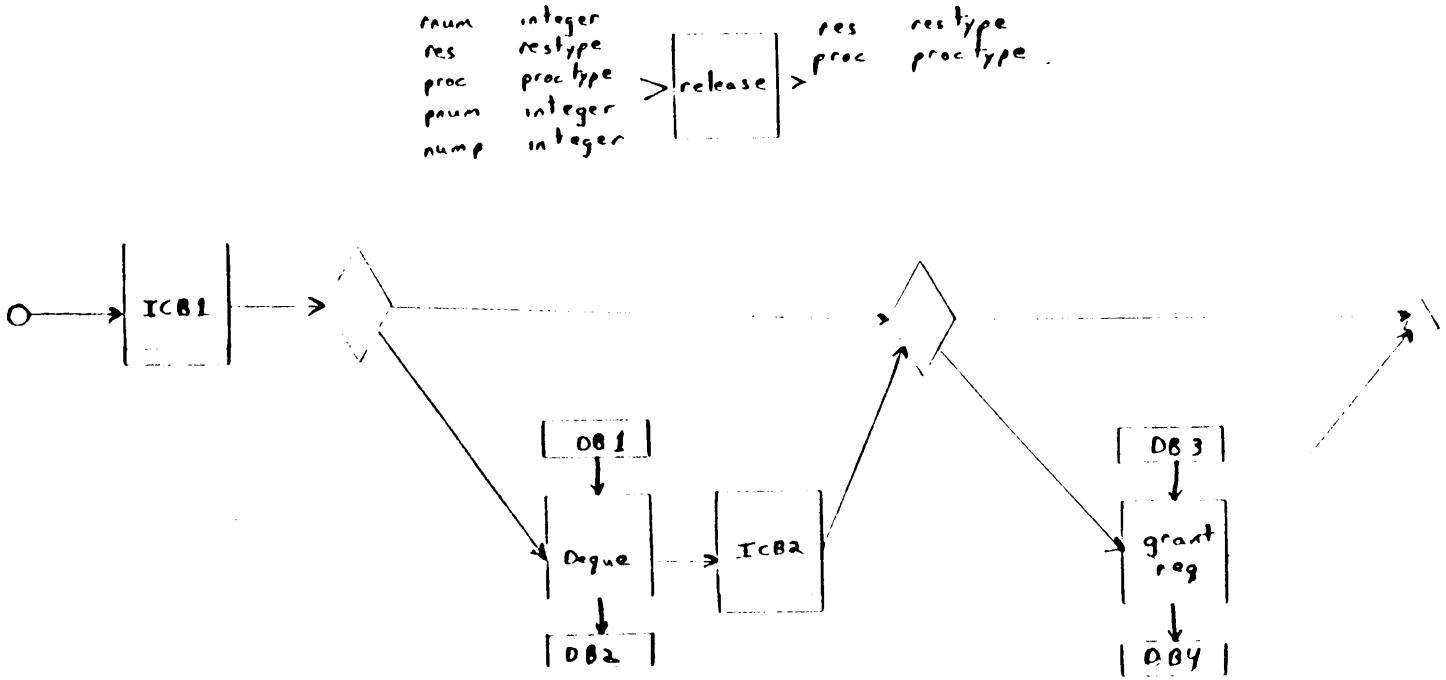
```
proc[pnum].alloc := proc[pnum].alloc - rnum
```

ICB5

```
proc[pnum].resources[TAPES].allocated := true;
```

ICB6

```
proc[pnum].req := 0;
res[TAPES].avail := res[TAPES].avail - rnum;
```



ICB1

```
res[TAPES].avail := res[TAPES].avail + rnum;
proc[pnum].alloc := proc[pnum].alloc - rnum;
```

ICB2

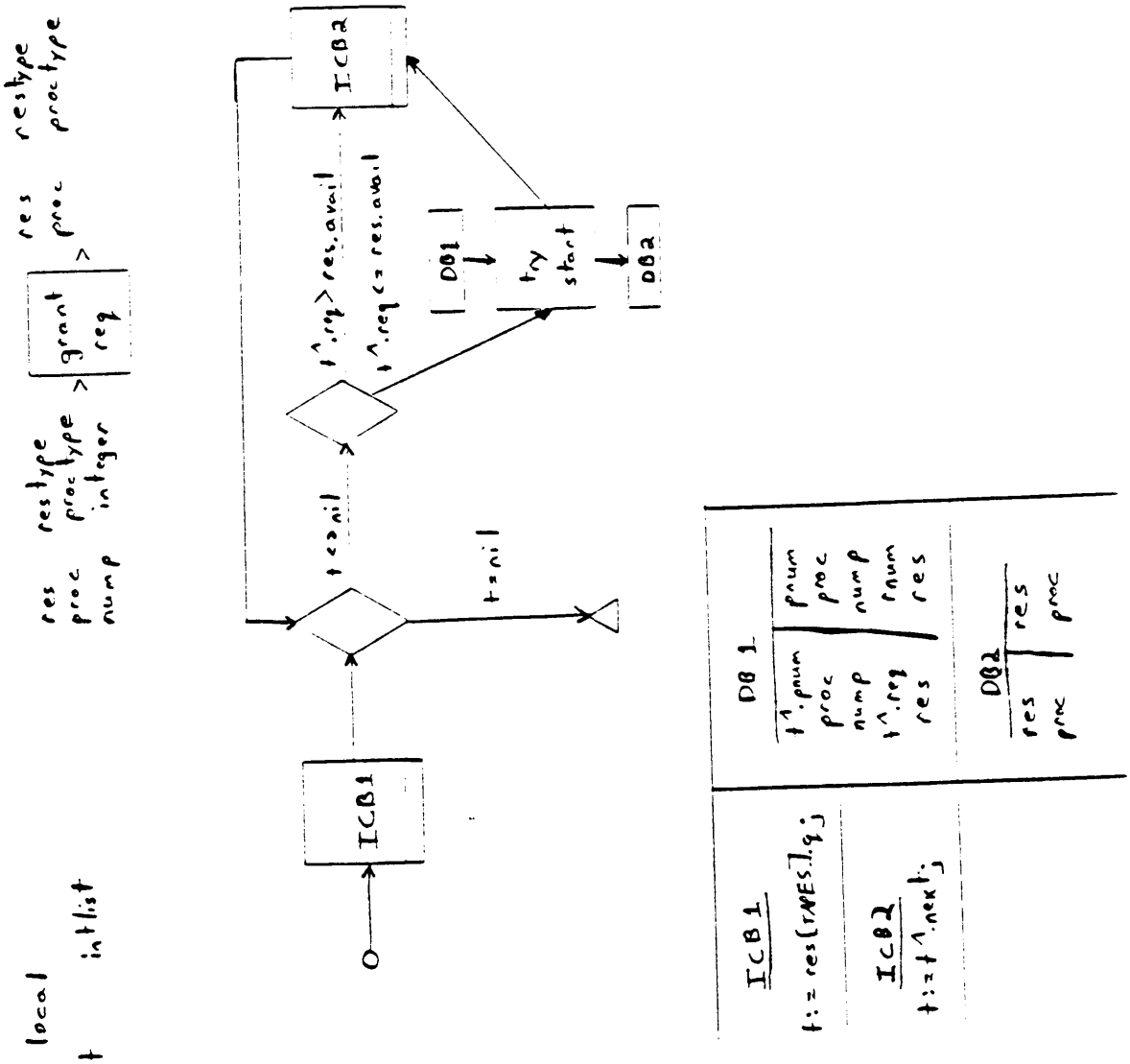
```
proc[pnum].resources[TAPES].allocated := false;
```

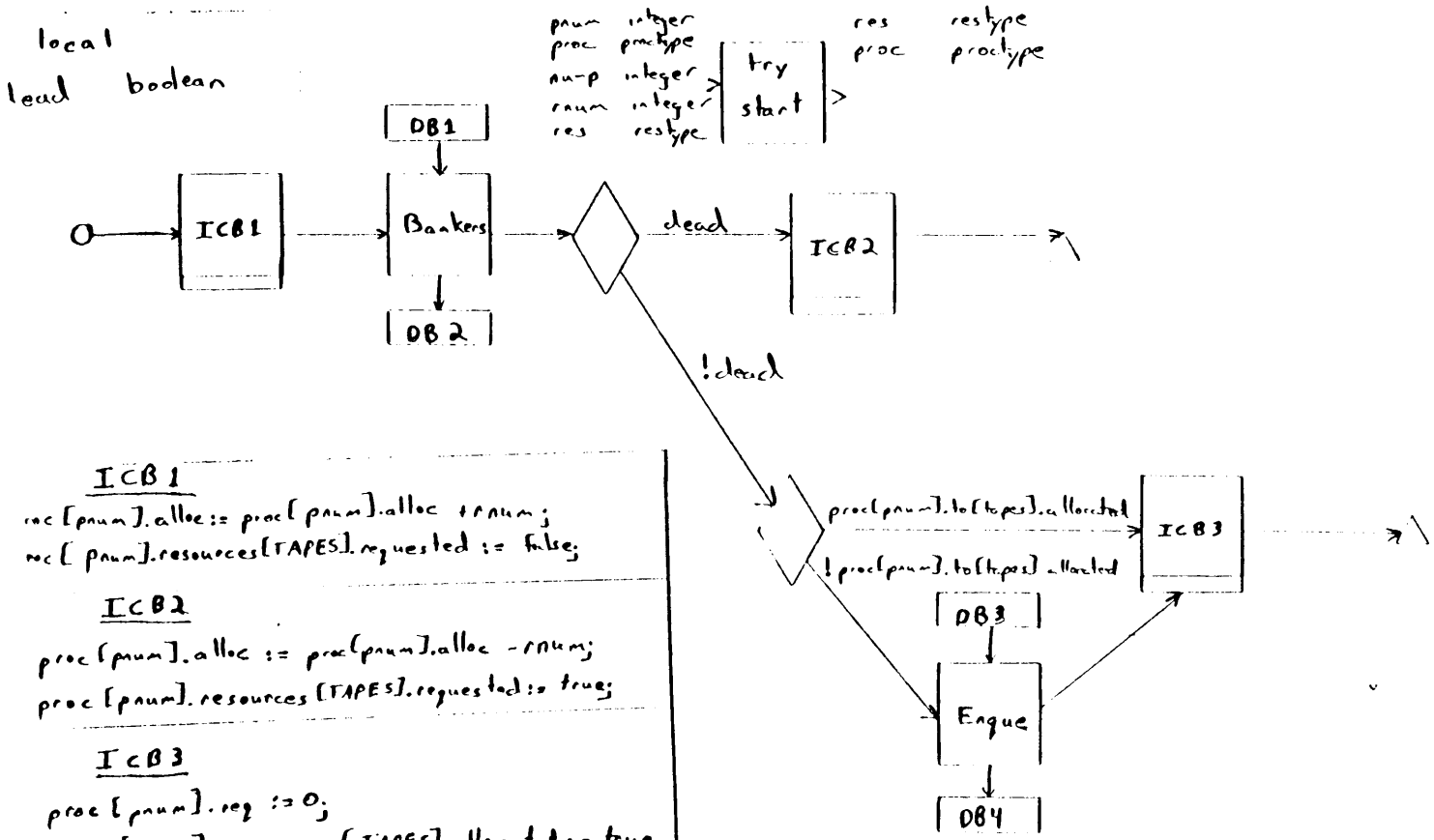
DB1	
rnum	pnum
res.toarc	q
res.toarctail	tail

DB2	
q	res.toarc
tail	res.toarctail

DB3	
res	res
proc	proc
nump	nump

DB4	
res	res
proc	proc


local
t intlist
res proc nump
restype proctype integer
grant req
res proc
restype proctype
ICB1
t <> nil
t = nil
t^.req > res.avail
t^.req <= res.avail
DB1
try start
DB2
ICB2
ICB1
t := res[RNES].q;
ICB2
t := t^.next;
DB 1
t^.pnum proc nump
t^.req res
pnum proc nump rnum res
DB2
res proc
res proc



ICB 1

```
proc[pnum].alloc := proc[pnum].alloc + rnum;
proc[pnum].resources[TAPES].requested := false;
```

ICB 2

```
proc[pnum].alloc := proc[pnum].alloc - rnum;
proc[pnum].resources[TAPES].requested := true;
```

ICB 3

```
proc[pnum].req := 0;
proc[pnum].resources[TAPES].allocated := true;
res[TAPES].avail := res[TAPES].avail - rnum;
```

DB 1

proc	proc
nump	nump

DB 2

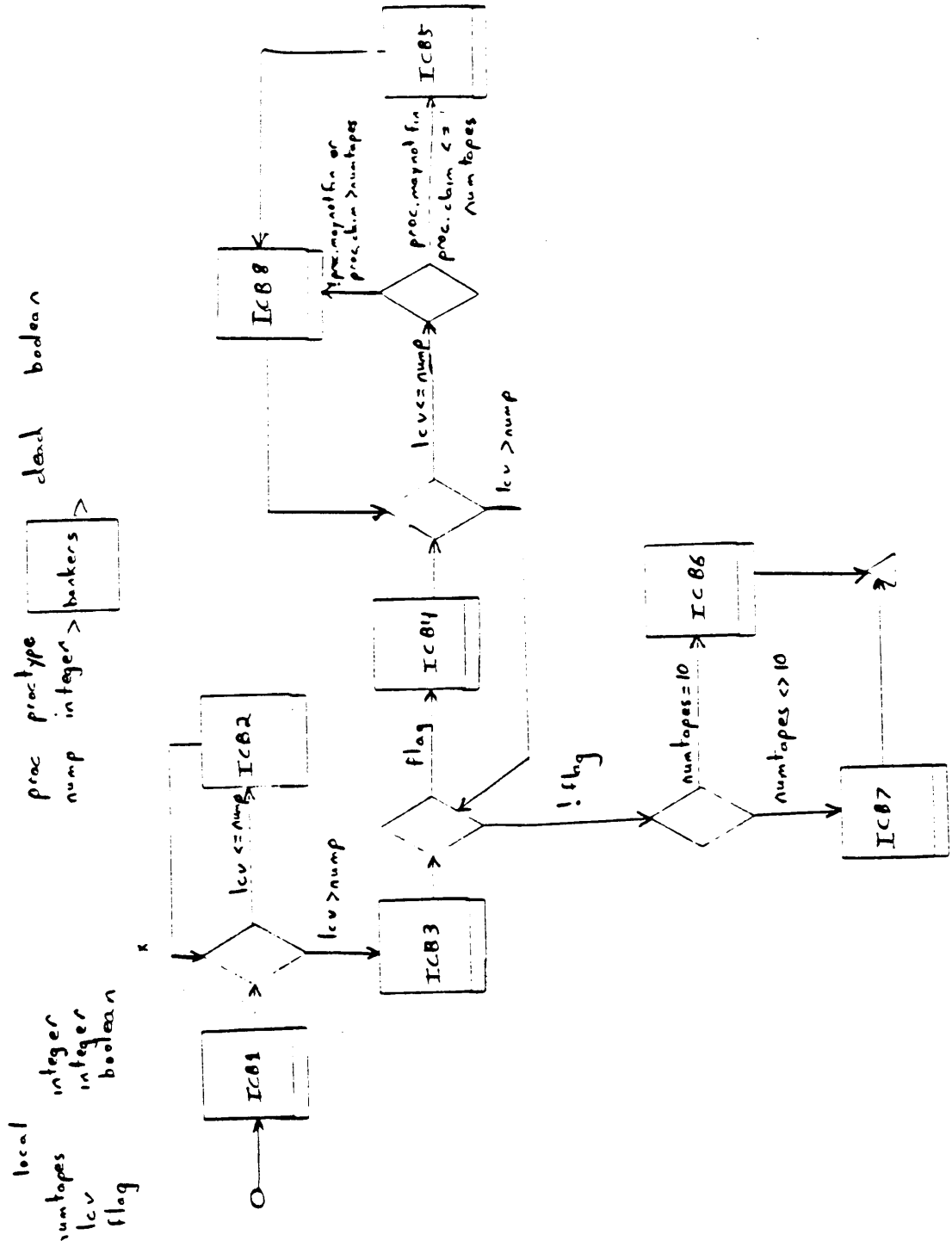
dead	dead

DB 3

pnum	pnum
rnum	rnum
res.toarc	q
res.toarctail	tail

DB 4

q	res.toarc
tail	res.toarctail


local
numtapes integer
lcv integer
flag boolean
proc proctype
nump integer
bankers
decl boolean
ICB1
ICB2
lcv <= nump
lcv > nump
ICB3
flag
!flag
ICB11
lcv <= nump
lcv > nump
ICB8
proc.maynot fin or proc.chim > numtapes
proc.maynot fin proc.chim <= numtapes
ICB5
numtapes = 10
numtapes <> 10
ICB6
ICB7

ICB 1

```
numtapes = 10;
```

ICB2

```
numtapes := numtapes - proc.alloc;
proc[kv].mayact fin := true;
proc[kv].claim := proc[kv].demand - proc[kv].alloc;
kv := kv+1;
```

ICB3

```
flag := true;
```

ICB4

```
flag := false;
kv := 1;
```

ICB5

```
proc[kv].mayact fin := false;
numtapes := numtapes + proc.alloc;
flag := true;
```

ICB6

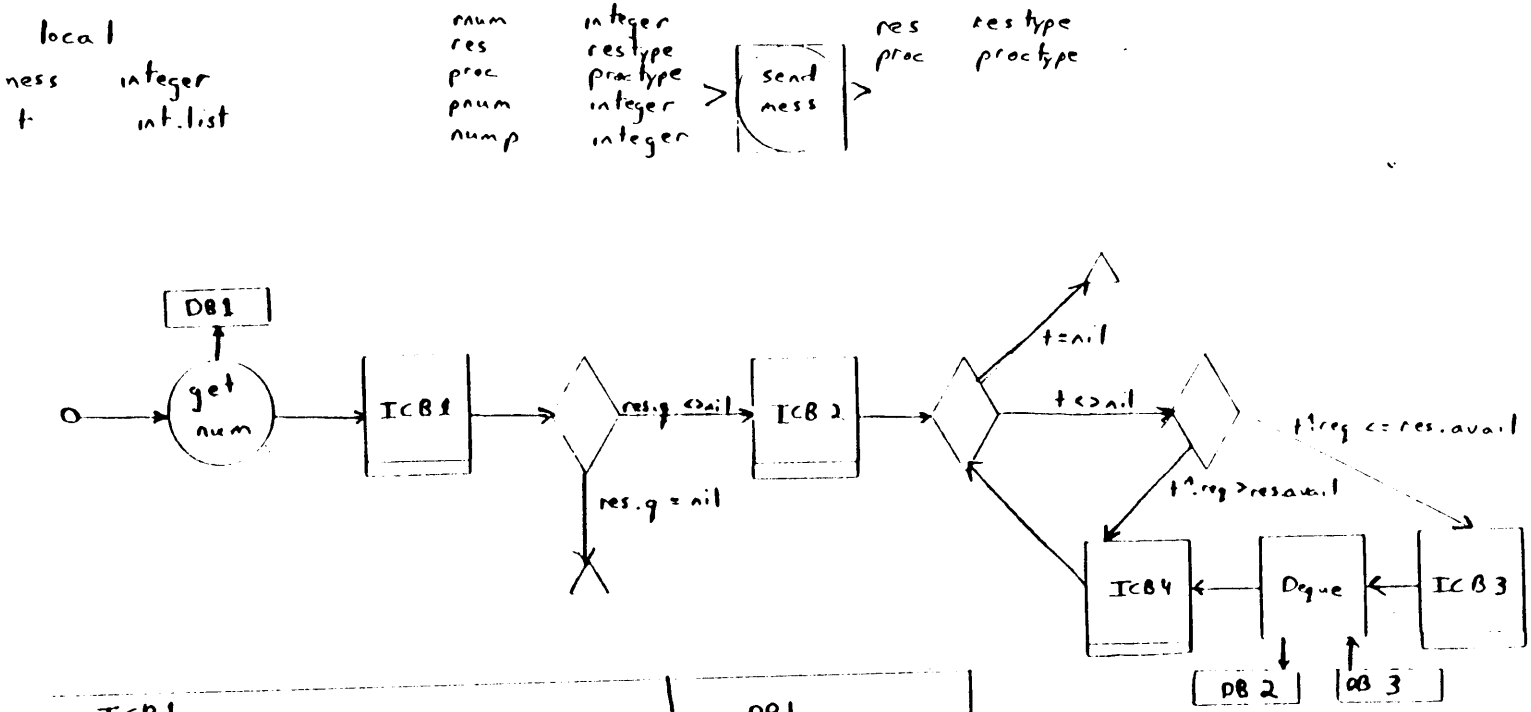
```
bankers := false;
```

ICB7

```
bankers := true;
```

ICB8

```
kv := kv + 1;
```



ICB1

```
res[rnum].avail := res[rnum].avail + mess;
```

ICB2

```
t := res[rnum].q;
```

ICB 3

```
res[rnum].avail := res[rnum].avail - t^.req;
proc [t^.pnum].resources [rnum].requested := false;
proc [t^.pnum].req := 0;
```

ICB4

```
t := t^.next;
```

DB1

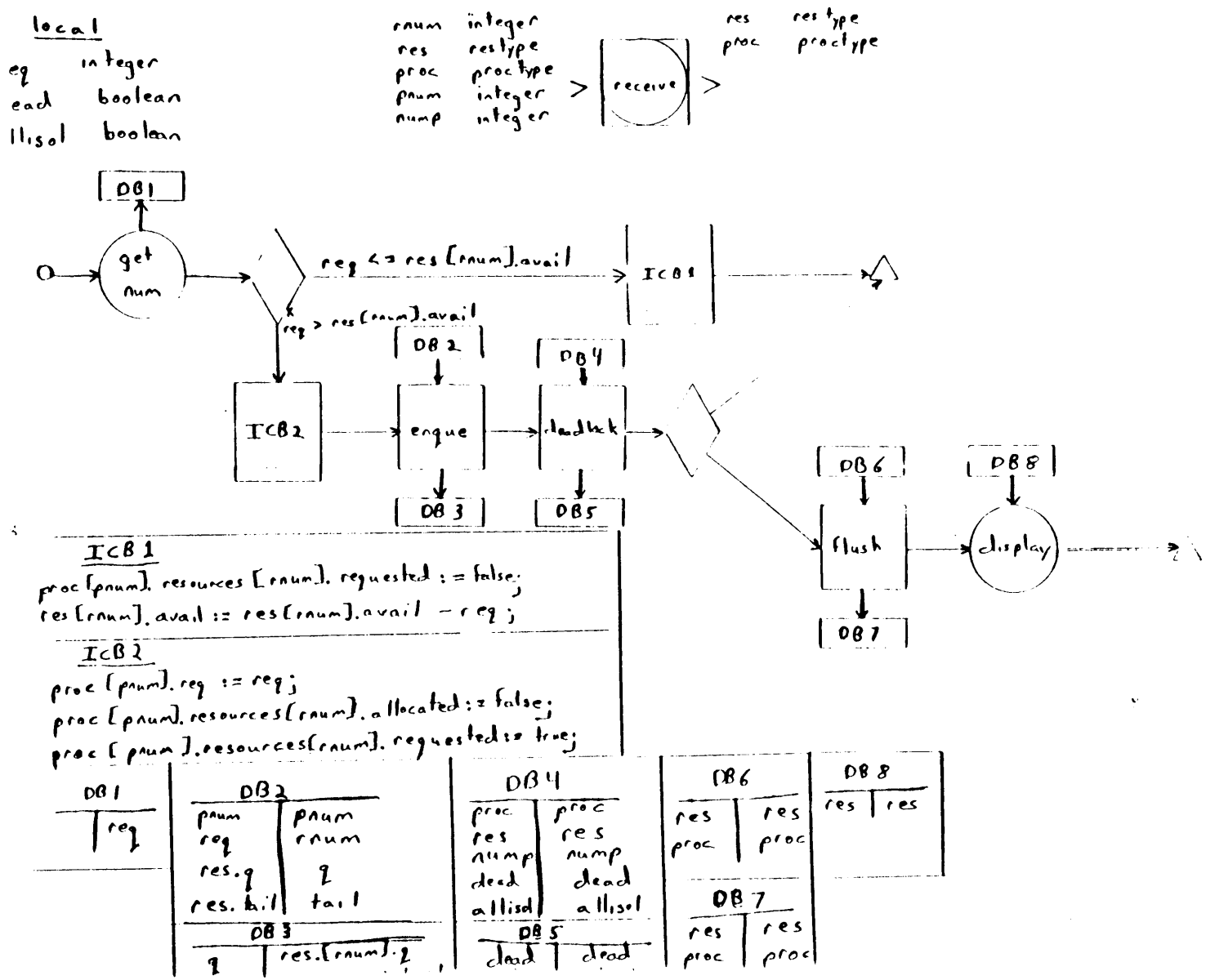
mess

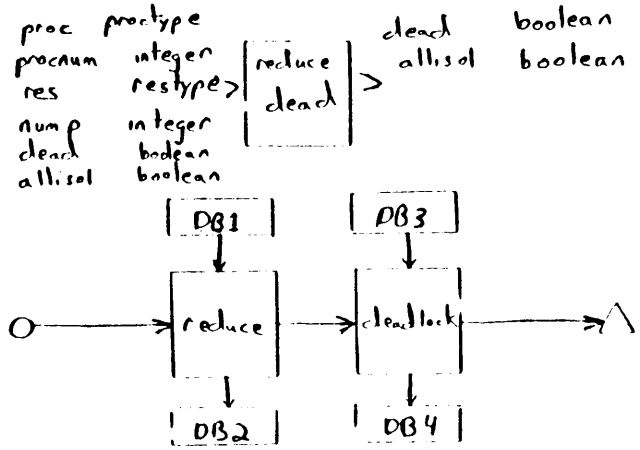
DB 2

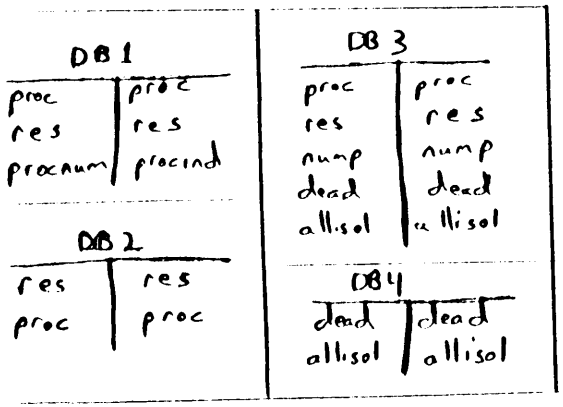
q	res[rnum].q
tail	res[rnum].tail

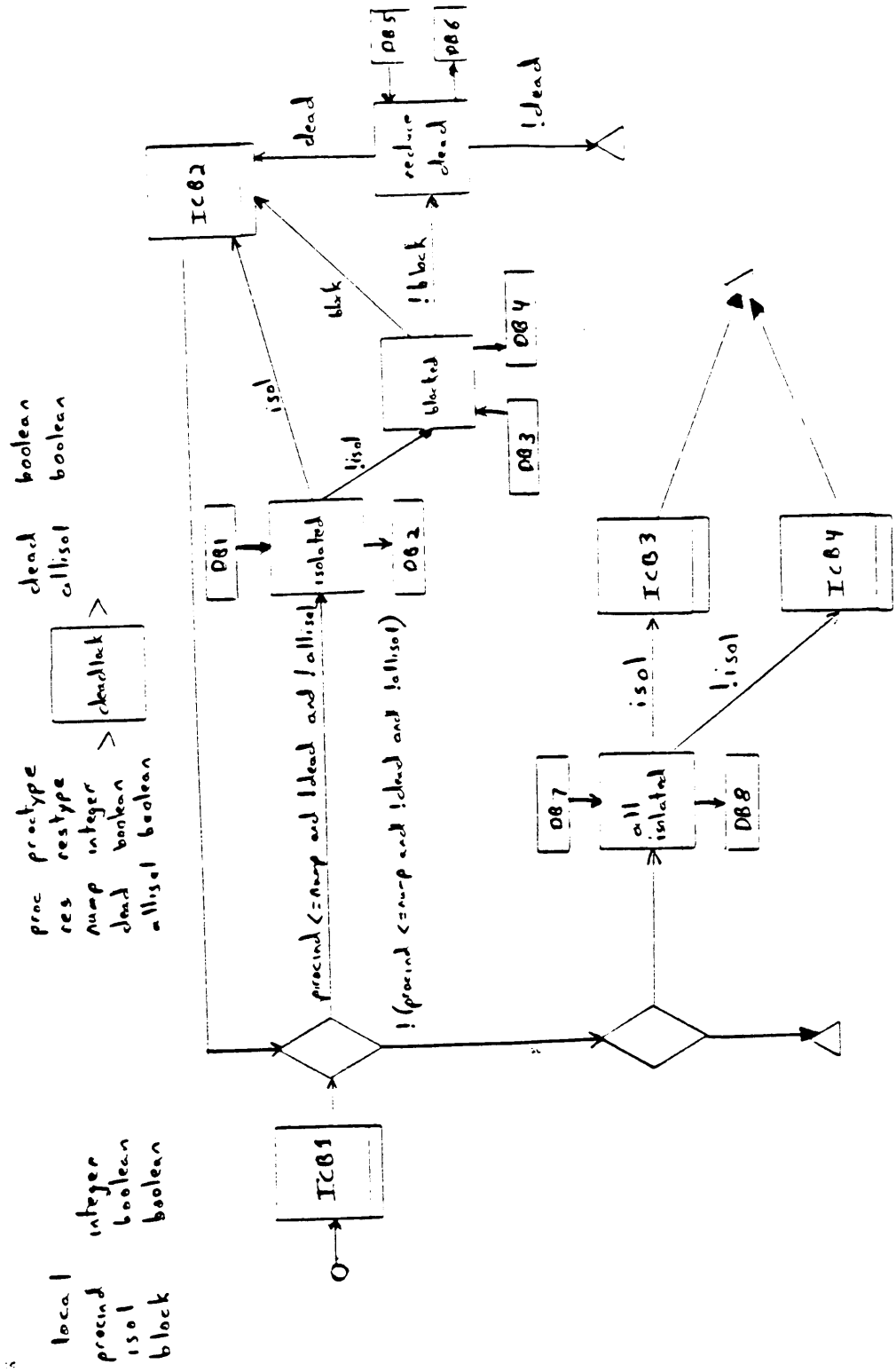
DB3

t^.pnum	pnum
res[rnum].q	q
res[rnum].tail	tail


local
eq integer
ead boolean
llisol boolean
rnum integer
res restype
proc proctype
pnum integer
nump integer
receive
res restype
proc proctype
DB1
get num
req <= res[rnum].avail
req > res[rnum].avail
ICB1
DB2
DB4
ICB2
enque
deadlock
DB3
DB5
DB6
DB8
flush
display
DB7
ICB1
proc[pnum].resources[rnum].requested := false;
res[rnum].avail := res[rnum].avail - req;
ICB2
proc[pnum].req := req;
proc[pnum].resources[rnum].allocated := false;
proc[pnum].resources[rnum].requested := true;
DB1
req
DB2
pnum
req
res.q
res.tail
pnum
rnum
q
tail
DB3
q
res.[rnum].q
DB4
proc
res
nump
dead
allisol
proc
res
nump
dead
allisol
DB5
dead
dead
DB6
res
proc
res
proc
DB7
res
proc
res
proc
DB8
res
res


proc proctype
procnum integer
res restype
nump integer
dead boolean
allisol boolean
reduce
dead
dead boolean
allisol boolean
DB1
DB3
reduce
deadlock
DB2
DB4


DB 1
proc proc
res res
procnum procind
DB 2
res res
proc proc
DB 3
proc proc
res res
nump nump
dead dead
allisol allisol
DB4
dead dead
allisol allisol


local
procind integer
isol boolean
block boolean
proc protype
res restype
nump integer
dead boolean
allisol boolean
deadblock
dead
allisol
ICB1
DB1
isolated
DB2
procind <=nump and !dead and !allisol
!(procind <=nump and !dead and !allisol)
ICB2
isol
!isol
blocked
DB3
DB4
block
!block
reduce dead
DB5
DB6
!dead
DB7
all isolated
DB8
ICB3
ICB4
isol
!isol

ICB1

```
procind := 1;
dead := false;
allisol := false;
```

ICB2

```
procind := procind +1;
```

ICB3

```
dead := false;
```

ICB4

```
dead := true;
```

DB1

oc	proc
ocind	procind

DB2

solated	isol

DB3

proc	proc
procind	procind
res	res

DB4

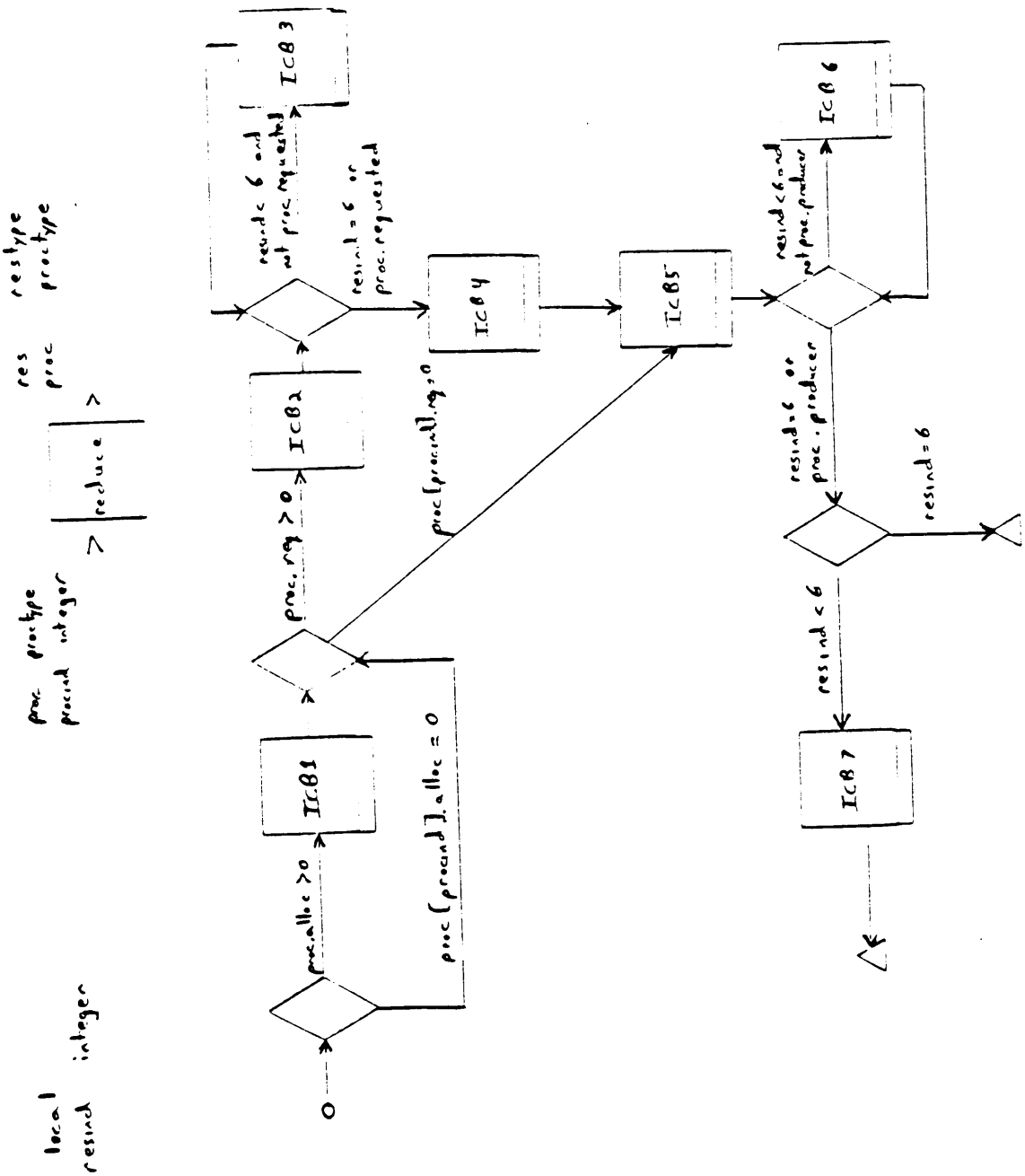
blocked	block

DB5

roc	proc
ocind	procind
es	res
ump	ump
lead	dead
llisol	allisol

DB6

dead	dead
allisol	allisol


local
resind integer
proc proctype
procind integer
reduce
res restype
proc proctype
proc.alloc > 0
ICB1
proc.req > 0
ICB2
resind < 6 and not proc.requested
ICB3
resind = 6 or proc.requested
ICB4
proc[procind].req = 0
proc[procind].alloc = 0
ICB5
resind < 6 and not proc.producer
ICB6
resind = 6 or proc.producer
resind = 6
resind < 6
ICB7

ICB 1

```
res[TAPES].avail := res[TAPES].avail + proc[procind].alloc;
proc[procind].alloc := 0;
```

ICB 2

```
resind := 0;
```

ICB 3

```
resind := resind + 1;
```

ICB 4

```
res[resind].avail := res[resind].avail - proc[procind].req;
proc[procind].req := 0;
proc[procind].resources[resind].requested := false;
```

ICB 5

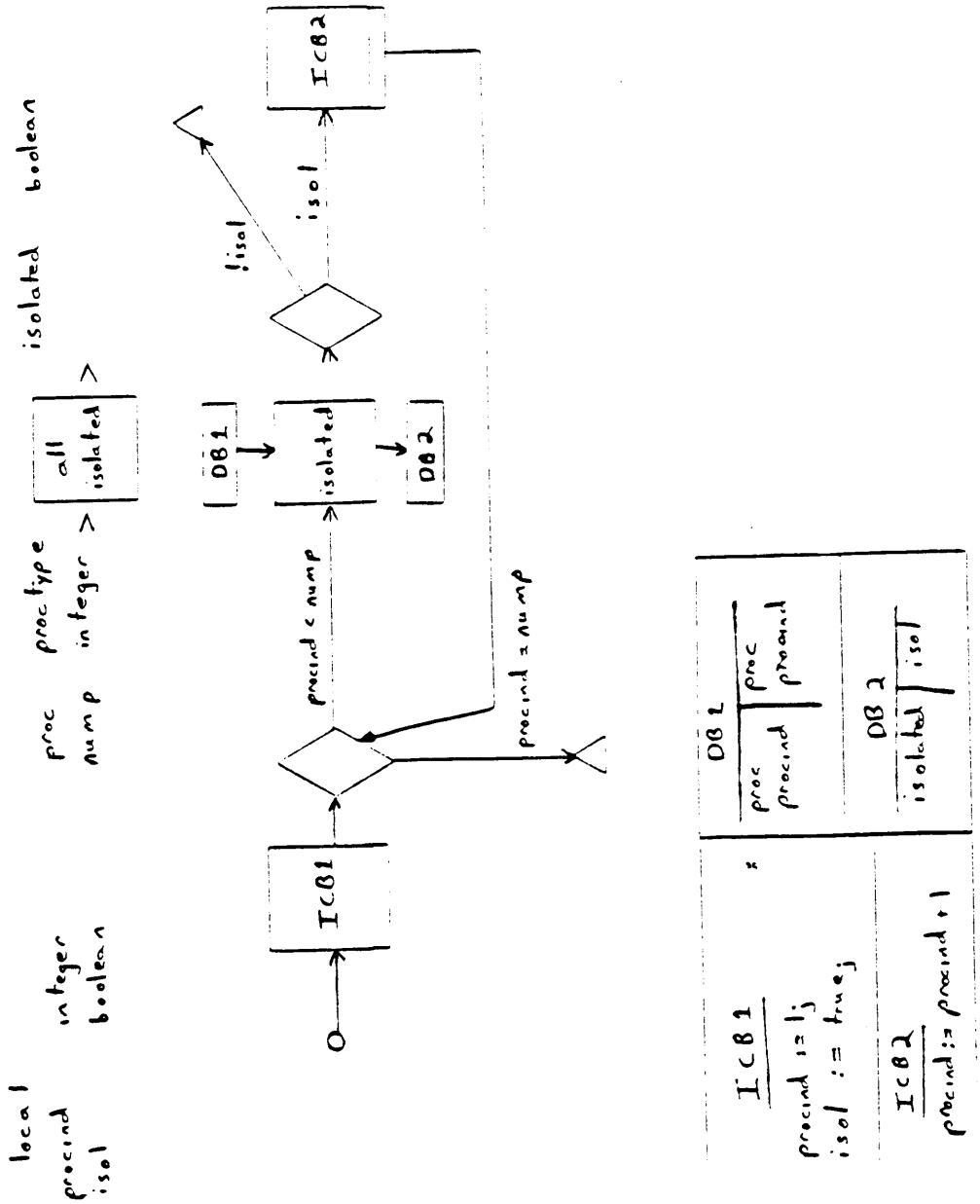
```
resind := 0;
```

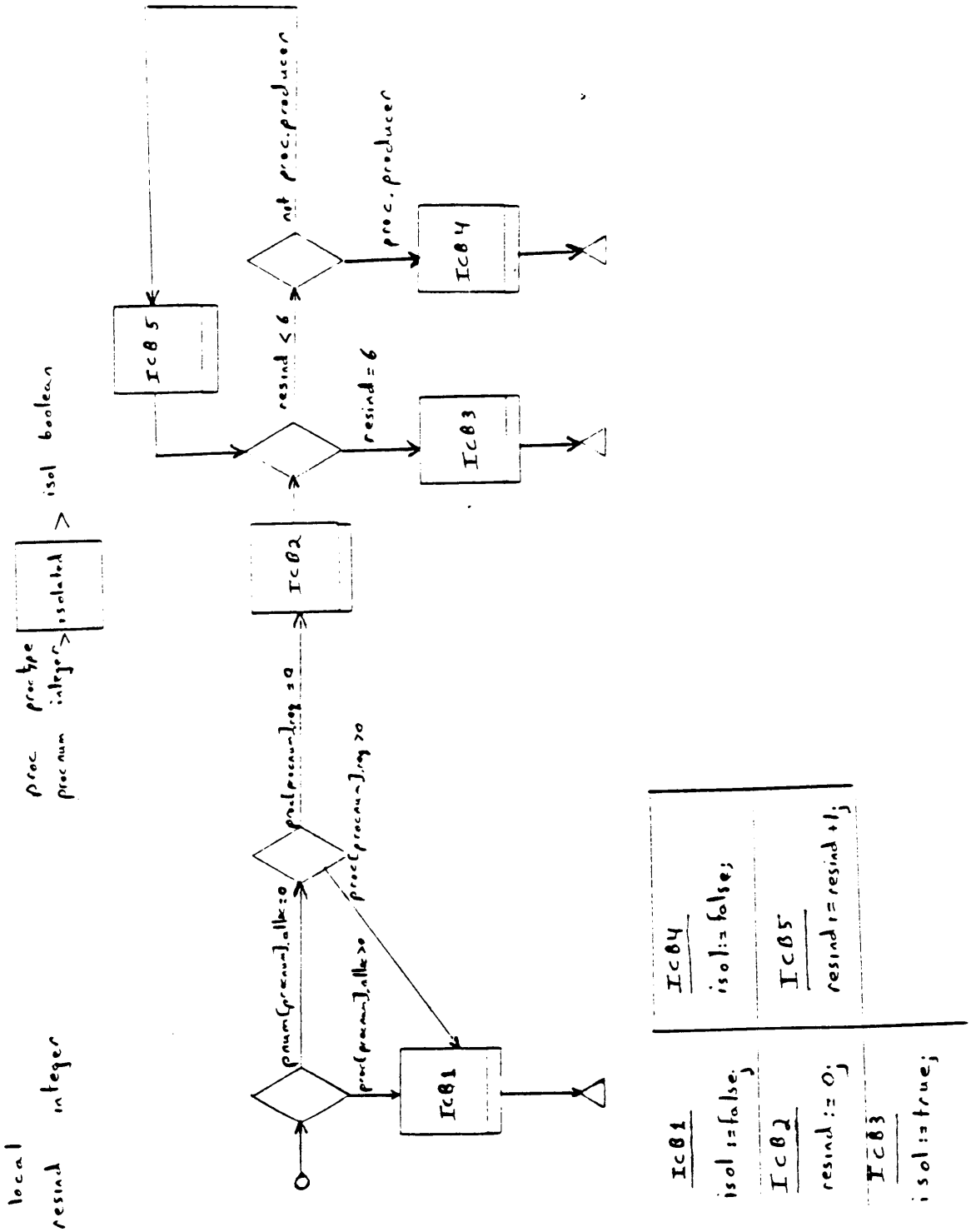
ICB 6

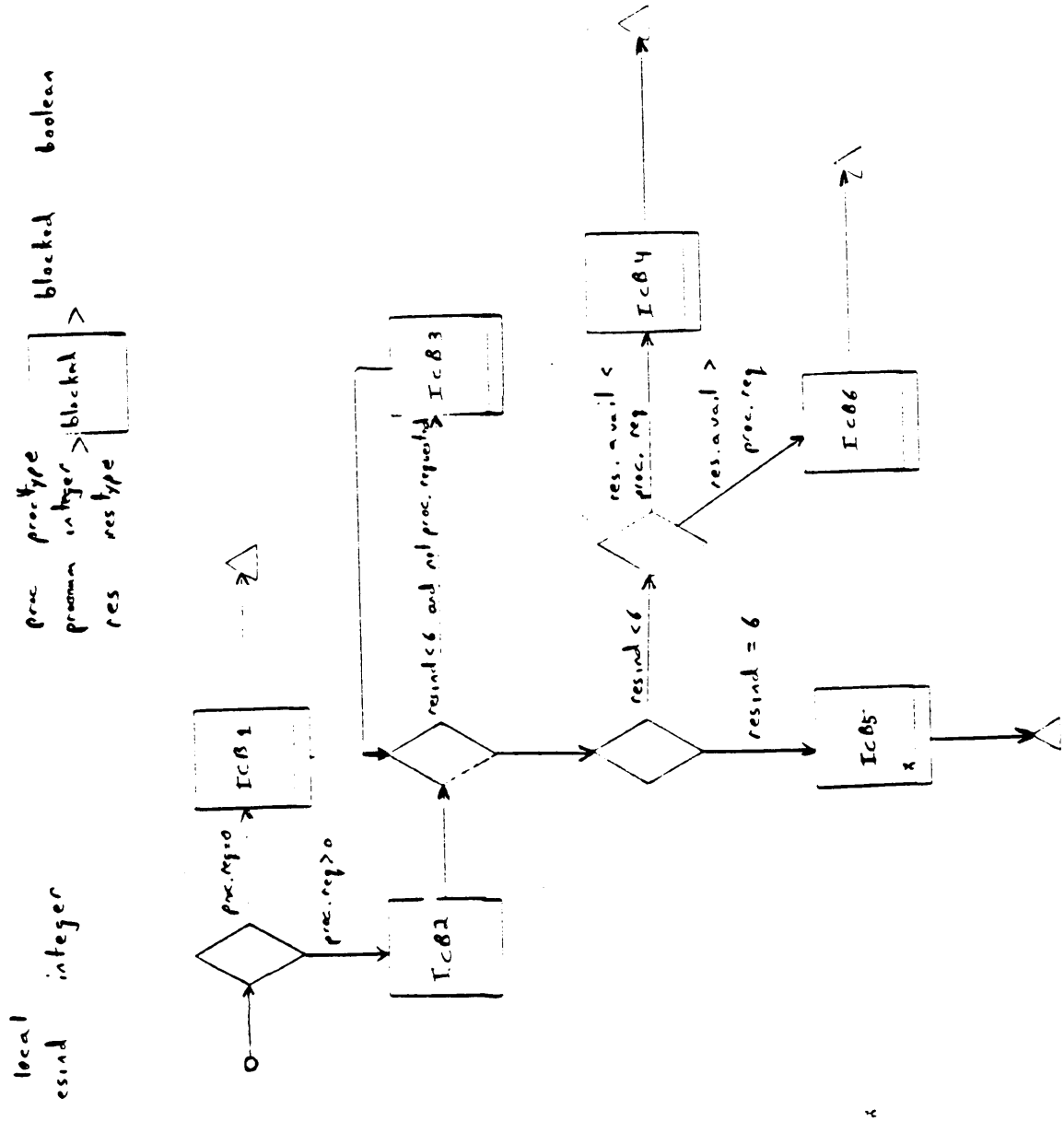
```
resind := resind + 1;
```

ICB 7

```
proc[procind].resources[resind].producer := false;
res[resind].avail := 1000000;
```


local
procind integer
isol boolean
proc proc type
nump integer
all
isolated
isolated boolean
ICB1
procind < nump
procind = nump
DB1
isolated
DB2
!isol
isol
ICB2
DB 1
proc procind
proc procand
DB 2
isolated isol
ICB 1
procind := 1;
isol := true;
ICB2
procind := procind + 1


local
resind integer
proc proc type
procnum integer
isolated
isol boolean
pnum[procnum].alle=0
proc[procnum].alle>0
pnal[procnum].reg=0
proc[procnum].reg>0
resind<6
resind=6
not proc.producer
proc.producer
ICB1
ICB2
ICB3
ICB4
ICB5
ICB1
isol:=false;
ICB2
resind:=0;
ICB3
isol:=true;
ICB4
isol:=false;
ICB5
resind:=resind+1;


local
resind integer
proc proctype
proman integer
res restype
blocked boolean
blocked
proc.req=0
proc.req>0
ICB1
ICB2
resind<6 and not proc.required
ICB3
resind<6
res. avail < proc. req
res. avail > proc. req
ICB4
ICB6
resind = 6
ICB5

ICB1

blocked := false;

ICB2

resind := 1;

ICB3

resind := resind + 1;

ICB4

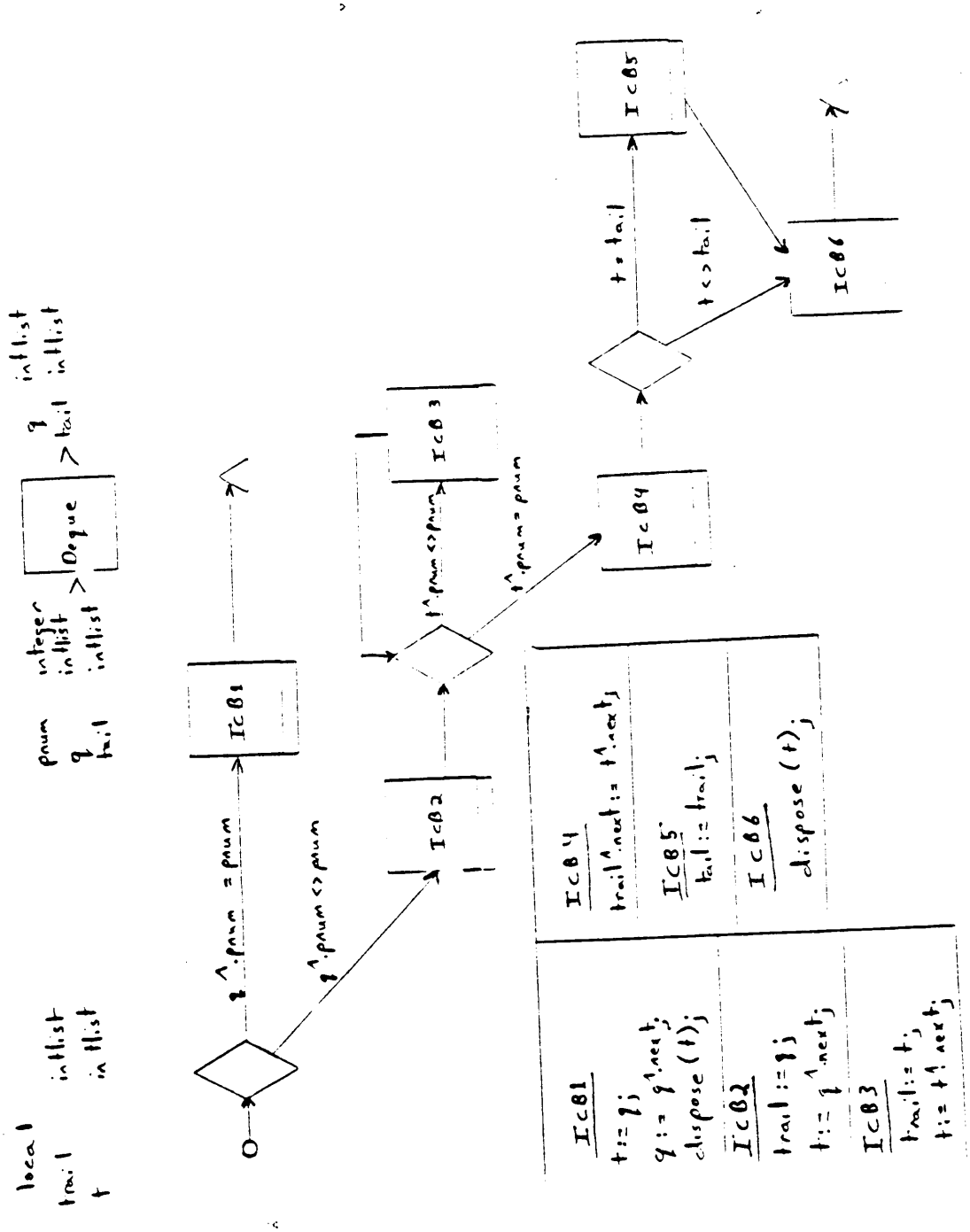
blocked := true;

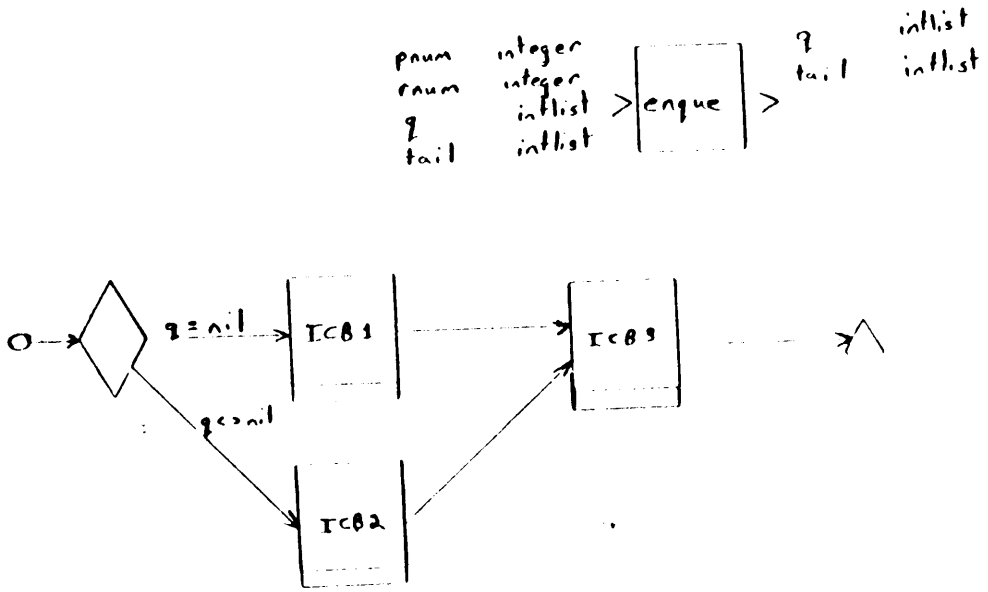
ICB5

blocked := false;

ICB6

blocked := false;


local
trail inlist
t inlist
pnum integer
q inlist
trail inlist
Queue
q inlist
trail inlist
IcB1
IcB2
IcB3
IcB4
IcB5
IcB6
q^.pnum = pnum
q^.pnum <> pnum
t^.pnum <> pnum
t^.pnum = pnum
t = trail
t <> trail
IcB1
t := q;
q := q^.next;
dispose (t);
IcB2
trail := q;
t := q^.next;
IcB3
trail := t;
t := t^.next;
IcB4
trail^.next := t^.next;
IcB5
trail := trail;
IcB6
dispose (t);



ICB1

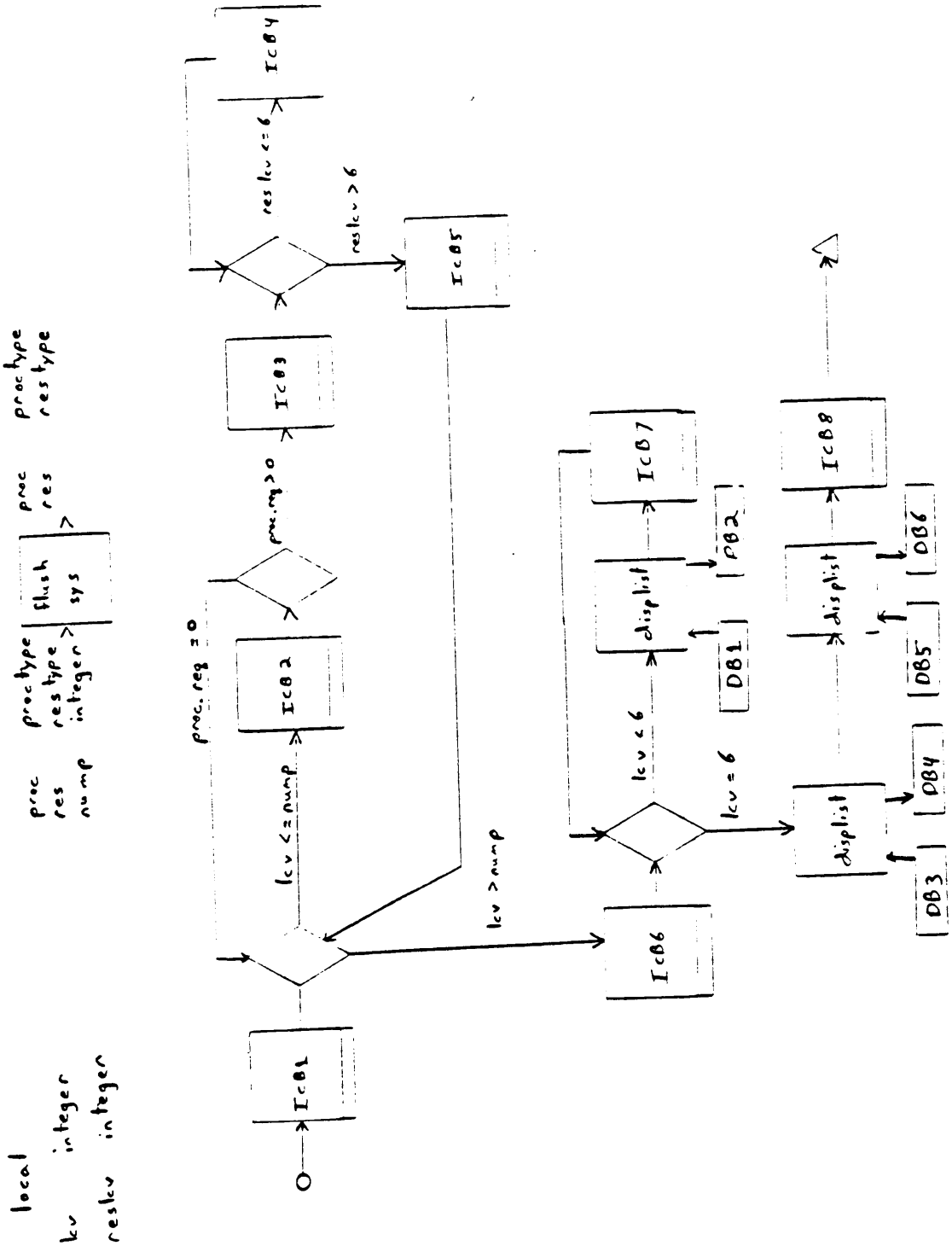
```
new(q);
tail := q
```

ICB2

```
new(tail^.next);
tail := tail^.next;
```

ICB3

```
tail^.pnum := pnum;
tail^.req  := rnum;
tail^.next := nil;
```


local
kv integer
reslev integer
proc
res
nump
proc type
res type
integer
flush
sys
proc
res
proc type
res type
ICB1
ICB2
ICB3
ICB4
ICB5
ICB6
ICB7
ICB8
proc. req = 0
proc. req > 0
res lev <= 6
res lev > 6
lev <= nump
lev > nump
lev < 6
lev = 6
displist
displist
displist
DB1
DB2
DB3
DB4
DB5
DB6

ICB1

```
lcv := 1;
```

ICB2

```
proc[lcv].alloc := 0;
proc[lcv].resources[TAPES].allocated := false;
```

ICB3

```
reslcv := 1;
```

ICB4

```
reslcv := reslcv + 1;
```

ICB5

```
proc[lcv].resources[reslcv].requested := false;
proc[lcv].req := 0;
```

ICB6

```
lcv := 1;
```

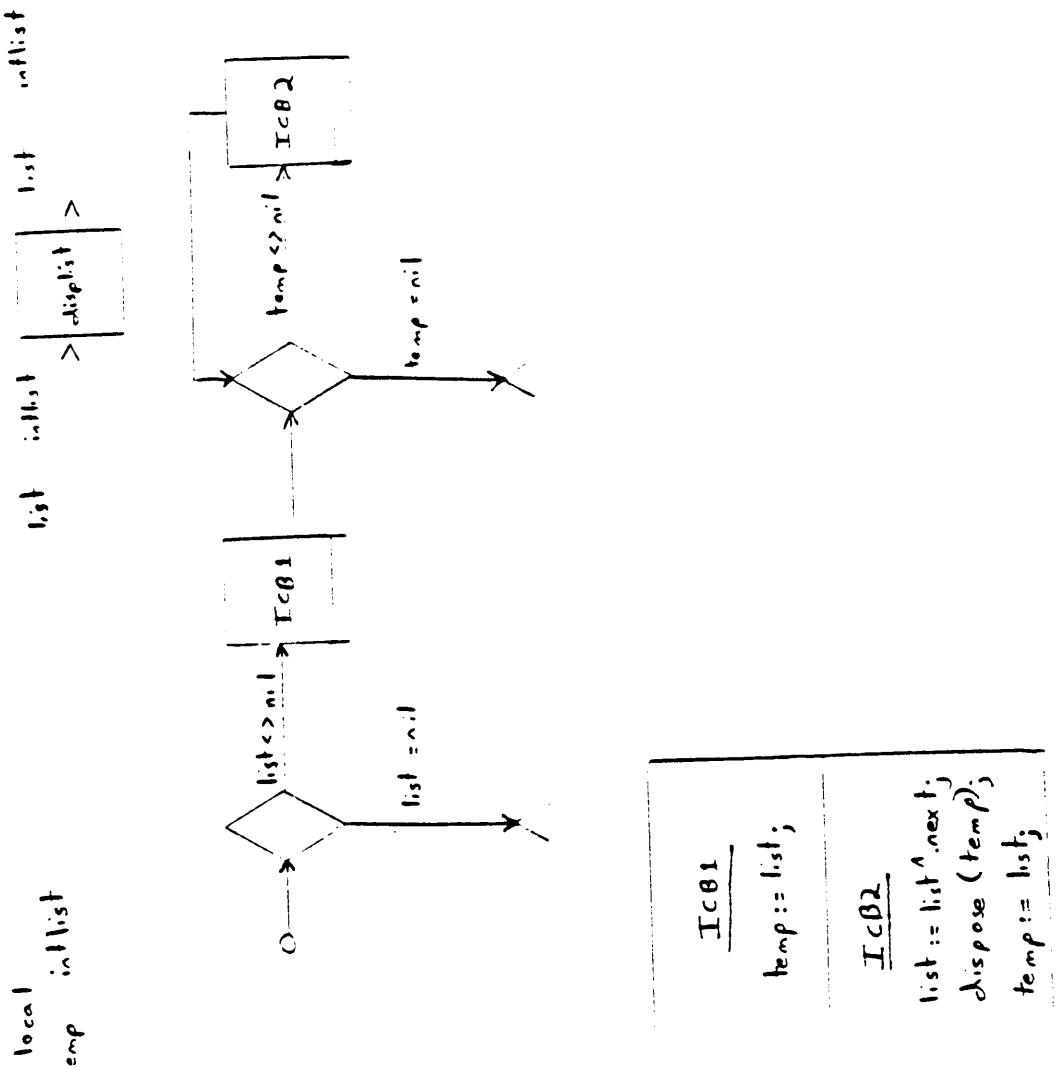
ICB7

```
res[lcv].tail := nil;
res[lcv].avail := 0;
```

ICB8

```
res[6].tail := nil;
res[6].toarctail := nil;
res[6].avail := 10;
```

DB1 res[lcv].q	**DB3** res[6].q	**DB5** res[6].toarc
DB2 res[lcv].q	**DB4** res[6].q	**DB6** res[6].toarc


local
temp intlist
list intlist
displist
list intlist
list <> nil
ICB1
list = nil
temp <> nil
ICB2
temp = nil
ICB1
temp := list;
ICB2
list := list^.next;
dispose (temp);
temp := list;

Appendix B. Sample Generated Relation Language Code

```
const true, TRUE;
const false, FALSE;
const not, NOT;
const eof, EOF;
const eoln, EOLN;
struct INPUT;
struct OUTPUT;
```

```
procedure   displist ( list )

begin
local   temp;



  cond (
        ( list & nil )
       & ( list & nil )
       )

    begin
    temp = list;
    end;



  cond (
        ( temp & nil )
       & ( temp & nil )
       & ( list & nil )
       & ( list & nil )
       )

    begin
    list = list;
    temp = list;
    end;
```

```
end;
```

```
procedure   flushsys ( proc, res, nump )

begin
local   lcv;
local   reslcv;


    begin
    lcv = 100;
    end;


  cond (
        ( lcv & nump )
      & ( lcv & nump )
      & ( proc & 100 )
      & ( proc & 100 )
      & ( reslcv & 100 )
      & ( reslcv & 100 )
      )

    begin
    proc = 100;
    proc = 100;
    end;


  cond (
```

```
    ( proc & 100 )
   & ( proc & 100 )
   & ( lcv & nump )
   & ( lcv & nump )
   & ( reslcv & 100 )
   & ( reslcv & 100 )
   )

  begin
  reslcv = 100;
  end;


cond (
    ( reslcv & 100 )
   & ( reslcv & 100 )
   & ( proc & 100 )
   & ( proc & 100 )
   & ( lcv & nump )
   & ( lcv & nump )
   )

  begin
  reslcv = reslcv & 100;
  end;


cond (
```

```
    ( proc & 100 )
   & ( proc & 100 )
   & ( lcv & nump )
   & ( lcv & nump )
   )

  begin
  proc = 100;
  proc = 100;
  end;

  begin
  lcv = 100;
  end;


cond (
    ( lcv & 100 )
   & ( lcv & 100 )
   & ( lcv & nump )
   & ( lcv & nump )
   & ( proc & 100 )
   & ( proc & 100 )
   & ( reslcv & 100 )
   & ( reslcv & 100 )
   )

  begin
```

```
displist ( res )
end;


cond (
    ( lcv & 100 )
  & ( lcv & 100 )
  & ( lcv & nump )
  & ( lcv & nump )
  & ( proc & 100 )
  & ( proc & 100 )
  & ( reslcv & 100 )
  & ( reslcv & 100 )
  )

begin
res = nil;
res = 100;
end;

begin
displist ( res )
end;

begin
displist ( res )
end;
```

```
    begin
    res = nil;
    res = nil;
    res = 100;
    end;


end;
```

```
procedure   enque ( pnum, rnum, q, tail )

begin



  cond (
        ( q & nil )
       & ( q & nil )
       )

    begin
    tail = q;
    end;

    begin
    tail = pnum;
    tail = rnum;
    tail = nil;
    end;



  cond (
        ( q & nil )
       & ( q & nil )
       )

    begin
```

```
        tail = 100;
        tail = tail;
        end;


    end;
```

```
procedure   deque ( pnum, q, tail )

begin
local   trail;
local   t;



  cond (
        ( q & pnum )
      & ( q & pnum )
      )

    begin
    t = q;
    q = q;
    t = nil;
    end;



  cond (
        ( q & pnum )
      & ( q & pnum )
      )

    begin
    trail = q;
    t = q;
```

```
  end;


cond (
      ( t & pnum )
    & ( t & pnum )
    & ( q & pnum )
    & ( q & pnum )
    )

  begin
  trail = t;
  t = t;
  end;


cond (
      ( q & pnum )
    & ( q & pnum )
    )

  begin
  trail = t;
  end;


cond (
      ( t & tail )
```

```
      & ( t & tail )
      & ( t & pnum )
      & ( t & pnum )
      & ( q & pnum )
      & ( q & pnum )
      )

    begin
    tail = trail;
    end;


  cond (
        ( q & pnum )
      & ( q & pnum )
      )

    begin
    t = nil;
    end;


end;
```

```
procedure   blocked ( proc, procnum, res, blocked )

begin
local   resind;



  cond (
        ( proc & 100 )
       & ( proc & 100 )
       )

    begin
    blocked = 100;
    end;



  cond (
        ( proc & 100 )
       & ( proc & 100 )
       )

    begin
    resind = 100;
    end;



  cond (
```

```
    ( resind & 100 & not & proc )
   & ( not & resind & 100 & not & proc )
   & ( proc & 100 )
   & ( proc & 100 )
   )

  begin
  resind = resind & 100;
  end;


cond (
    ( res & proc )
   & ( res & proc )
   & ( resind & 100 )
   & ( resind & 100 )
   & ( resind & 100 & not & proc )
   & ( not & resind & 100 & not & proc )
   & ( proc & 100 )
   & ( proc & 100 )
   )

  begin
  blocked = 100;
  end;


cond (
```

```
    ( resind & 100 )
   & ( resind & 100 )
   & ( resind & 100 & not & proc )
   & ( not & resind & 100 & not & proc )
   & ( proc & 100 )
   & ( proc & 100 )
   )

  begin
  blocked = 100;
  end;


cond (
    ( res & proc )
   & ( res & proc )
   & ( resind & 100 )
   & ( resind & 100 )
   & ( resind & 100 & not & proc )
   & ( not & resind & 100 & not & proc )
   & ( proc & 100 )
   & ( proc & 100 )
   )

  begin
  blocked = 100;
  end;
```

```
end;
```

```
procedure   isolated ( proc, procnum, isol )

begin
local   resind;



  cond (
        ( proc & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
      )

    begin
    resind = 100;
    end;



  cond (
        ( proc )
      & ( not & proc )
      & ( resind & 100 )
      & ( resind & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
```

```
    )

  begin
  resind = resind & 100;
  end;


cond (
      ( proc & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    )

  begin
  isol = 100;
  end;


cond (
      ( resind & 100 )
    & ( resind & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( proc )
    & ( not & proc )
```

```
        )

        begin
        isol = 100;
        end;


      cond (
            ( proc )
          & ( not & proc )
          & ( resind & 100 )
          & ( resind & 100 )
          & ( proc & 100 )
          & ( proc & 100 )
          & ( proc & 100 )
          & ( proc & 100 )
          )

        begin
        isol = 100;
        end;


    end;
```

```
procedure   allisolated ( proc, nump, isol )

begin
local   procind;


    begin
    procind = 100;
    isol = 100;
    end;


  cond (
        ( procind & nump )
      & ( procind & nump )
      & ( not & isol )
      & ( isol )
      )

    begin
    isolated ( proc, procind, isol )
    end;


  cond (
        ( not & isol )
      & ( isol )
      & ( procind & nump )
```

```
    & ( procind & nump )
    )

  begin
  procind = procind & 100;
  end;


end;
```

```
procedure   reduce ( procind, proc, res )

begin
local   resind;



  cond (
        ( proc & 100 )
      & ( proc & 100 )
      )

    begin
    res = res & proc;
    proc = 100;
    end;



  cond (
        ( proc & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
      & ( proc & 100 )
      )

    begin
    resind = 100;
    end;
```

```
cond (
    ( resind & 100 & not & proc )
   & ( resind & 100 & proc )
   & ( proc & 100 )
   & ( proc & 100 )
   & ( proc & 100 )
   & ( proc & 100 )
   )

  begin
  resind = resind & 100;
  end;


cond (
    ( proc & 100 )
   & ( proc & 100 )
   & ( proc & 100 )
   & ( proc & 100 )
   )

  begin
  res = res & proc;
  proc = 100;
  proc = 100;
  end;
```

```
  begin
  resind = 100;
  end;


cond (
     ( resind & 100 )
    & ( resind & 100 )
    & ( resind & 100 & not & proc )
    & ( resind & 100 & proc )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( proc & 100 )
    & ( resind & 100 & not & proc )
    & ( resind & 100 & proc )
    )

  begin
  proc = 100;
  res = 100;
  end;


cond (
     ( resind & 100 & not & proc )
    & ( resind & 100 & proc )
```

```
        & ( proc & 100 )
        & ( proc & 100 )
        & ( proc & 100 )
        & ( proc & 100 )
        & ( resind & 100 & not & proc )
        & ( resind & 100 & proc )
        )

      begin
      resind = resind & 100;
      end;


end;
```

```
extern     reducedead;
procedure   deadlock ( proc, res, nump, dead, allisol )

begin
local   procind;
local   isol;
local   block;


    begin
    procind = 100;
    dead = 100;
    allisol = 100;
    end;


  cond (
        ( procind & nump & not & dead & not & allisol )
      & ( procind & nump & dead & allisol )
      & ( block )
      & ( not & block )
      & ( dead )
      & ( not & dead )
      )

    begin
    isolated ( proc, procind, isol )
    end;
```

```
cond (
    ( isol )
   & ( not & isol )
   & ( procind & nump & not & dead & not & allisol )
   & ( procind & nump & dead & allisol )
   & ( block )
   & ( not & block )
   & ( dead )
   & ( not & dead )
   )

  begin
  procind = procind & 100;
  end;


cond (
    ( isol )
   & ( not & isol )
   & ( procind & nump & not & dead & not & allisol )
   & ( procind & nump & dead & allisol )
   & ( dead )
   & ( not & dead )
   )

  begin
```

```
  blocked ( proc, procind, res, block )
  end;


cond (
     ( block )
    & ( not & block )
    & ( isol )
    & ( not & isol )
    & ( procind & nump & not & dead & not & allisol )
    & ( procind & nump & dead & allisol )
    )

  begin
  reducedead ( proc, procind, res, nump, dead, allisol )
  end;


cond (
     ( allisol )
    & ( not & allisol )
    & ( procind & nump & not & dead & not & allisol )
    & ( procind & nump & dead & allisol )
    & ( isol )
    & ( not & isol )
    & ( block )
    & ( not & block )
    & ( dead )
```

```
    & ( not & dead )
    )

  begin
  allisolated ( proc, nump, allisol )
  end;


cond (
     ( allisol )
    & ( not & allisol )
    & ( allisol )
    & ( not & allisol )
    & ( procind & nump & not & dead & not & allisol )
    & ( procind & nump & dead & allisol )
    & ( isol )
    & ( not & isol )
    & ( block )
    & ( not & block )
    & ( dead )
    & ( not & dead )
    )

  begin
  dead = 100;
  end;
```

```
cond (
      ( allisol )
     & ( not & allisol )
     & ( allisol )
     & ( not & allisol )
     & ( procind & nump & not & dead & not & allisol )
     & ( procind & nump & dead & allisol )
     & ( isol )
     & ( not & isol )
     & ( block )
     & ( not & block )
     & ( dead )
     & ( not & dead )
     )

   begin
   dead = 100;
   end;


end;
```

```
procedure   reducedead ( proc, procnum, res, nump, dead, allisol )

begin


    begin
    reduce ( procnum, proc, res )
    end;

    begin
    deadlock ( proc, res, nump, dead, allisol )
    end;


end;
```

```
procedure   receive ( res, proc, rnum, pnum, nump )

begin
local   req;
local   dead;
local   allisol;


    begin
    req = INPUT;
    end;


  cond (
        ( req & res )
      & ( req & res )
      )

    begin
    proc = 100;
    res = res & req;
    end;


  cond (
        ( req & res )
      & ( req & res )
      )
```

```
  begin
  proc = req;
  proc = 100;
  proc = 100;
  end;


cond (
     ( req & res )
    & ( req & res )
    )

  begin
  enque ( pnum, req. res, res )
  end;


cond (
     ( req & res )
    & ( req & res )
    )

  begin
  deadlock ( proc, res, nump, dead, allisol )
  end;
```

```
cond (
      ( not & dead )
    & ( dead )
    & ( req & res )
    & ( req & res )
    )

  begin
  flushsys ( res, proc, nump )
  end;


cond (
      ( not & dead )
    & ( dead )
    & ( req & res )
    & ( req & res )
    )

  begin
  OUTPUT = res;
  end;


end;
```

```
procedure   sendmess ( res, proc, mum )

begin
local   mess;
local   t;


    begin
    mess = INPUT;
    end;

    begin
    res = res & mess;
    end;


  cond (
        ( res & nil )
      & ( res & nil )
      )

    begin
    t = res;
    end;


  cond (
        ( t & nil )
```

```
    & ( t & nil )
    & ( res & nil )
    & ( res & nil )
    )

  begin
  t = t;
  end;


cond (
      ( t & res )
    & ( t & res )
    & ( t & nil )
    & ( t & nil )
    & ( res & nil )
    & ( res & nil )
    )

  begin
  deque ( t, res, res )
  end;


cond (
      ( t & res )
    & ( t & res )
    & ( t & nil )
```

```
    & ( t & nil )
    & ( res & nil )
    & ( res & nil )
    )

  begin
  res = res & t;
  proc = 100;
  proc = 100;
  end;


end;
```

```
procedure   bankers ( proc, nump, dead )

begin
local   numtapes;
local   lcv;
local   flag;


    begin
    numtapes = 100;
    end;


  cond (
        ( lcv & nump )
       & ( lcv & nump )
       )

    begin
    numtapes = numtapes & proc;
    proc = 100;
    proc = proc & proc;
    lcv = lcv & 100;
    end;


  cond (
        ( lcv & nump )
```

```
    & ( lcv & nump )
    & ( flag )
    & ( not & flag )
    & ( lcv & nump )
    & ( lcv & nump )
    )

  begin
  lcv = lcv & 100;
  end;

  begin
  flag = 100;
  end;


cond (
     ( flag )
    & ( not & flag )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( not & proc & proc & numtapes )
    & ( proc & proc & numtapes )
    )

  begin
```

```
flag = 100;
lcv = 100;
end;


cond (
    ( not & proc & proc & numtapes )
  & ( proc & proc & numtapes )
  & ( lcv & nump )
  & ( lcv & nump )
  & ( flag )
  & ( not & flag )
  & ( lcv & nump )
  & ( lcv & nump )
  )

begin
proc = 100;
numtapes = numtapes & proc;
flag = 100;
end;


cond (
    ( numtapes & 100 )
  & ( numtapes & 100 )
  & ( flag )
  & ( not & flag )
```

```
    & ( lcv & nump )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( not & proc & proc & numtapes )
    & ( proc & proc & numtapes )
    )

  begin
  dead = 100;
  end;


cond (
      ( numtapes & 100 )
    & ( numtapes & 100 )
    & ( flag )
    & ( not & flag )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( lcv & nump )
    & ( not & proc & proc & numtapes )
    & ( proc & proc & numtapes )
    )

  begin
  dead = 100;
```

```
    end;


end;
```

```
procedure   trystart ( pnum, proc, nump, rnum, res )

begin
local   dead;


    begin
    proc = proc & rnum;
    proc = 100;
    end;

    begin
    bankers ( proc, nump, dead )
    end;


  cond (
       ( dead )
      & ( not & dead )
      )

    begin
    proc = proc & rnum;
    proc = 100;
    end;


  cond (
```

```
      ( dead )
     & ( not & dead )
     )

    begin
    proc = 100;
    proc = 100;
    res = res & rnum;
    end;


  cond (
      ( not & proc )
     & ( proc )
     & ( dead )
     & ( not & dead )
     )

    begin
    enque ( pnum, rnum, res, res )
    end;


end;
```

```
procedure   grantreq ( res, proc, nump )

begin
local   t;


    begin
    t = res;
    end;


  cond (
        ( t & nil )
      & ( t & nil )
      )

    begin
    t = t;
    end;


  cond (
        ( t & res )
      & ( t & res )
      & ( t & nil )
      & ( t & nil )
      )
```

```
    begin
    trystart ( t, proc, nump, t, res )
    end;


end;
```

```
procedure   release ( res, proc, rnum, pnum, nump )

begin


    begin
    res = res & rnum;
    proc = proc & rnum;
    end;


  cond (
        ( proc & 100 )
      & ( proc & 100 )
      )

    begin
    deque ( pnum, res, res )
    end;


  cond (
        ( proc & 100 )
      & ( proc & 100 )
      )

    begin
    proc = 100;
```

```
    end;


  cond (
        ( res & nil )
       & ( res & nil )
       & ( proc & 100 )
       & ( proc & 100 )
       )

    begin
    grantreq ( res, proc, nump )
    end;


end;
```

```
procedure   request ( res, proc, rnum, pnum, nump )

begin
local   dead;


    begin
    dead = 100;
    end;


  cond (
        ( res & rnum )
      & ( res & rnum )
      & ( dead )
      & ( not & dead )
      )

    begin
    proc = 100;
    proc = 100;
    end;


  cond (
        ( res & rnum )
      & ( res & rnum )
      & ( dead )
```

```
    & ( not & dead )
    )

  begin
  enque ( pnum, rnum, res, res )
  end;


cond (
      ( res & rnum )
    & ( res & rnum )
    )

  begin
  proc = proc & rnum;
  proc = 100;
  end;


cond (
      ( res & rnum )
    & ( res & rnum )
    )

  begin
  bankers ( proc, nump, dead )
  end;
```

```
cond (
      ( dead )
    & ( not & dead )
    & ( res & rnum )
    & ( res & rnum )
    )

  begin
  proc = proc & rnum;
  end;


cond (
      ( not & proc )
    & ( proc )
    & ( dead )
    & ( not & dead )
    & ( res & rnum )
    & ( res & rnum )
    )

  begin
  enque ( pnum, rnum, res, res )
  end;


cond (
```

```
      ( not & proc )
     & ( proc )
     & ( dead )
     & ( not & dead )
     & ( res & rnum )
     & ( res & rnum )
     )

    begin
    proc = 100;
    end;


  cond (
      ( dead )
     & ( not & dead )
     & ( res & rnum )
     & ( res & rnum )
     )

    begin
    proc = 100;
    res = res & rnum;
    end;


end;
```

```
procedure   skipblanks ( ch )

begin



  cond (
        ( ch & 100 )
       & ( ch & 100 )
       )

     begin
     ch = INPUT;
     end;



end;
```

```
procedure   processline ( res, proc, nump )

begin
local   ch;
local   pnum;
local   com;
local   rnum;


    begin
    skipblanks ( ch )
    end;


  cond (
        ( ch & 100 )
      & ( ch & 100 )
      )

    begin
    ch = INPUT;
    end;


  cond (
        ( ch & 100 )
      & ( ch & 100 )
      )
```

```
  begin
  pnum = INPUT;
  end;


cond (
     ( ch & 100 )
    & ( ch & 100 )
    )

  begin
  skipblanks ( com )
  end;


cond (
     ( ch & 100 )
    & ( ch & 100 )
    )

  begin
  ch = INPUT;
  end;


cond (
     ( ch & 100 )
```

```
    & ( ch & 100 )
    )

  begin
  rnum = INPUT;
  end;


cond (
      ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( ch & 100 )
    & ( ch & 100 )
    )

  begin
  request ( res, proc, rnum, pnum, nump )
  end;


cond (
      ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( ch & 100 )
```

```
    & ( ch & 100 )
    )

  begin
  release ( res, proc, rnum, pnum, nump )
  end;


cond (
      ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( ch & 100 )
    & ( ch & 100 )
    )

  begin
  sendmess ( res, proc, rnum )
  end;


cond (
      ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( com & 100 )
    & ( ch & 100 )
```

```
    & ( ch & 100 )
    )

  begin
  receive ( res, proc, rnum, pnum, nump )
  end;


cond (
      ( ch & 100 )
    & ( ch & 100 )
    )

  begin
  OUTPUT = res;
  end;


cond (
      ( ch & 100 )
    & ( ch & 100 )
    )

  begin
  OUTPUT = 100;
  end;
```

```
end;
```

```
procedure   simulate ( res, proc, nump )

begin



  cond (
        ( not & eof )
       & ( eof )
       )

     begin
     processline ( res, proc, nump )
     end;



end;
```

```
procedure   initprocess ( process, res, proc, nump )

begin
local   ch;
local   demand;
local   resource;


    begin
    skipblanks ( ch )
    end;

    begin
    ch = INPUT;
    end;

    begin
    demand = INPUT;
    end;


  cond (
        ( not & eoln )
      & ( eoln )
      )

    begin
    resource = INPUT;
```

```
    end;


  cond (
      ( not & eoln )
     & ( eoln )
     )

    begin
    res = 100;
    res = process;
    res = nil;
    proc = 100;
    proc = 100;
    proc = 100;
    end;

    begin
    proc = demand;
    proc = 100;
    nump = nump & 100;
    proc = 100;
    end;


end;
```

```
procedure   initsys ( res, proc, nump )

begin
local   processnum;
local   ch;


    begin
    processnum = 100;
    res = 100;
    end;

    begin
    skipblanks ( ch )
    end;

    begin
    ch = INPUT;
    end;

    begin
    processnum = INPUT;
    end;


  cond (
        ( processnum & 100 )
       & ( processnum & 100 )
```

```
    )

  begin
  initprocess ( processnum, res, proc, nump )
  end;


cond (
      ( processnum & 100 )
    & ( processnum & 100 )
    )

  begin
  skipblanks ( ch )
  end;


cond (
      ( processnum & 100 )
    & ( processnum & 100 )
    )

  begin
  ch = INPUT;
  end;


cond (
```

```
    ( processnum & 100 )
   & ( processnum & 100 )
   )

  begin
  processnum = INPUT;
  end;

  begin
  ch = INPUT;
  end;


end;
```

```
local   res;
local   proc;
local   nump;


    begin
    initsys ( res, proc, nump )
    end;

    begin
    simulate ( res, proc, nump )
    end;

null
```

Appendix C. Sample Pascal Source Code

```
program deadlock (input, output);

const

TAPES = 6;
MAXPROC = 10;
BLANK = ' ';
MAXTAPES = 10;

type

  intlist = ¬intrec;
  intrec = record
     pnum : integer;
     req  : integer;
```

```
    next : intlist;
     end;
resrec = record
    toarc : intlist;
    toarctail : intlist;
    avail : integer;
    q : intlist;
    tail : intlist;
     end;
restype = array [1..TAPES] of resrec;


procreslist = record
        requested : boolean;
        allocated : boolean;
        producer  : boolean;
      end;
reslist = array [1..TAPES] of procreslist;
procrec = record
    demand : integer;
    resources : reslist;
    alloc : integer;
    maynotfin : boolean;
    claim : integer;
    req : integer;
      end;
```

```
  proctype = array [1..MAXPROC] of procrec;



var

  res : restype;

  proc : proctype;

  nump : integer;
```

```
procedure printq ( q : intlist );

var
  t : intlist;



begin
  if ( q < > nil ) then
    begin
    t := q;
    writeln; writeln;
    writeln ('     Process Number          Units Requested' );
    while ( t < > nil ) do
      begin
      writeln ( '            ',t¬.pnum:1,
          '                    ', t¬.req:1 );
      t := t¬.next;
      end;
    end
  else
    begin
    writeln;
    writeln ( '     The Queue is Empty' );
    end;
end;
```

```
procedure display ( res : restype );


var lcv : integer;



begin


  writeln;
  for lcv:= 1 to TAPES do
    begin
    if ( lcv <> TAPES ) then
      begin
      writeln ( 'Message Area ', lcv:1, ' Queue' );
      writeln ( '====================' );
      end
    else
      begin
      writeln ( 'Tape Queue' );
      writeln ( '==========' );
      end;
    printq ( res[lcv].q );
    writeln; writeln; writeln; writeln;
    end;
end;
```

```
procedure displist ( var list : intlist );

var
  temp : intlist;


begin
  if (list < > nil) then
    begin
    temp := list;
    while (temp < > nil) do
      begin
      list := list¬.next;
      dispose ( temp );
      temp := list;
      end;
    end;
end;
```

```
procedure flushsys ( var proc : proctype; var res : restype;
                    nump : integer );

var
  lcv : integer;
  reslcv : integer;


begin
  for lcv := 1 to nump do
    begin
    proc[lcv].alloc := 0;
    proc[lcv].resources[TAPES].allocated := false;
    if ( proc[lcv].req > 0 ) then
      begin
      reslcv := 1;
      while ( (reslcv <= TAPES) and
            not proc[lcv].resources[reslcv].requested ) do
        reslcv := reslcv + 1;
      proc[lcv].resources[reslcv].requested := false;
      proc[lcv].req := 0;
      end;
    end;
  for lcv := 1 to (TAPES - 1) do
    begin
    displist ( res[lcv].q );
    res[lcv].tail := nil;
    res[lcv].avail := 0;
```

```
    end;
  displist ( res[TAPES].q );
  res[TAPES].tail := nil;
  displist ( res[TAPES].toarc );
  res[TAPES].toarctail := nil;
  res[TAPES].avail := MAXTAPES;
end;
```

```
procedure enque ( pnum, rnum : integer; var q, tail : intlist );



begin
  if (q = nil)
    then begin
    new ( q );
    tail := q;
    end
  else
    begin
    new (tail¬.next);
    tail := tail¬.next;
    end;
  tail¬.pnum := pnum;
  tail¬.req := rnum;
  tail¬.next := nil;
end;
```

```
procedure deque ( pnum : integer; var q, tail : intlist );

var
  trail, t : intlist;


begin
  if (q¬.pnum = pnum)
    then begin
    t := q;
    q := q¬.next;
    dispose ( t );
    end
  else
    begin
    trail := q;
    t := q¬.next;
    while ( t¬.pnum < > pnum ) do
      begin
      trail := t;
      t := t¬.next;
      end;
    trail¬.next := t¬.next;
    if (t = tail) then
      tail := trail;
    dispose ( t );
    end;
end;
```

```
function blocked ( proc : proctype; procnum: integer;
                res : restype ) : boolean;



var
  resind : integer;



begin
  if (proc[procnum].req = 0)
    then blocked := false
  else
    begin
    resind := 1;
    while ( (resind <= TAPES) and
      not (proc[procnum].resources[resind].requested)) do
      resind := resind + 1;
    if not (resind > TAPES) then
      begin
      if (res[resind].avail < proc[procnum].req) then
        blocked := true
      else
        blocked := false;
      end
    else
      blocked := false;
    end
end;
```

```
function isolated ( proc : proctype; procnum : integer )
            : boolean;

var
  resind : integer;


begin
  if (proc[procnum].alloc > 0)
    then
    isolated := false
  else
    if (proc[procnum].req > 0)
      then
      isolated := false
    else begin
      resind := 1;
      while ( (resind < TAPES) and
        not (proc[procnum].resources[resind].producer)) do
        resind := resind + 1;
      if (resind < TAPES)
        then isolated := false
      else
        isolated := true
     end
end;
```

```
function allisolated ( proc : proctype; nump : integer )
                : boolean;

var
  procind : integer;
  isol : boolean;


begin
  procind := 1;
  isol := true;
  while ( isol and (procind <= nump)) do
    begin
    isol := isolated ( proc, procind );
    procind := procind + 1;
    end;
  allisolated := isol;
end;
```

```
procedure reduce (var proc : proctype; var res : restype;
                procind : integer );



var
  resind : integer;



begin
  if (proc[procind].alloc > 0)
    then begin
    res[TAPES].avail := res[TAPES].avail + proc[procind].alloc;
    proc[procind].alloc := 0;
    end;
  if (proc[procind].req > 0)
    then begin
    resind := 0;
    while ( (resind < TAPES) and
      not (proc[procind].resources[resind].requested)) do
      resind := resind + 1;
    res[resind].avail := res[resind].avail - proc[procind].req;
    proc[procind].req := 0;
    proc[procind].resources[resind].requested := false;
    end;
  resind := 0;
  while ( (resind < TAPES) and
    not (proc[procind].resources[resind].producer)) do
    resind := resind + 1;
```

```
  if (resind < TAPES)
    then begin
    proc[procind].resources[resind].producer := false;
    res[resind].avail := 1000000;
    end;
end;
```

```
procedure reducedead (proc : proctype; procnum:integer;
                res : restype; nump : integer;
                var dead, allisol : boolean);forward;



procedure deadlock ( proc : proctype; res : restype;
                nump : integer; var dead, allisol : boolean);


var
  procind : integer;
  block : boolean;
  isol : boolean;



begin
   procind := 1;
   dead := false;
   allisol := false;
   while ((procind <= nump) and (not dead) and (not allisol)) do
     begin
     isol := isolated (proc, procind);
     if (not isol)
       then begin
       block := blocked (proc, procind, res);
       if (not block) then
         begin
         reducedead (proc, procind,res,nump, dead,
              allisol);
```

```
            end;
              end;
        procind := procind + 1;
        end;
      if (not allisol) then
        begin
        allisol := allisolated (proc, nump);
        if (allisol) then
          dead := false
        else
          begin
          dead := true;
          end;
        end;
    end;
```

```
procedure reducedead;



begin
  reduce (proc, res, procnum);
  deadlock (proc, res, nump, dead, allisol);
end;
```

```
procedure receivemess (var res : restype; var proc : proctype;
        rnum, pnum, nump : integer);

var
  req : integer;
  dead, allisol : boolean;


begin
  readln ( req );
  if (req < = res[rnum].avail)
    then begin
    proc[pnum].resources[rnum].requested := false;
    res[rnum].avail := res[rnum].avail - req;
    end
  else begin
    proc[pnum].req := req;
    proc[pnum].resources[rnum].allocated := false;
    proc[pnum].resources[rnum].requested := true;
    enque (pnum, req, res[rnum].q, res[rnum].tail);
    deadlock (proc, res, nump, dead, allisol);
    if (dead) then
      begin
      page(output);
      writeln;
      writeln ( 'The System is Deadlocked' );
      writeln;
      display ( res );
```

```
        flushsys( proc, res, nump );
        end;
      end;
end;
```

```
procedure sendmess (var res : restype; var proc : proctype;
                    rnum: integer);

var
  mess : integer;
  t : intlist;


begin
  readln ( mess );
  res[rnum].avail := res[rnum].avail + mess;
  if (res[rnum].q < > nil)
    then begin
    t := res[rnum].q;
    while (t < > nil) do
      begin
      if (t¬.req < = res[rnum].avail)
        then begin
        res[rnum].avail := res[rnum].avail - t¬.req;
        proc[t¬.pnum].resources[rnum].requested := false;
        proc[t¬.pnum].req := 0;
        deque ( t¬.pnum, res[rnum].q, res[rnum].tail );
        end;
      t := t¬.next;
      end;
    end;
end;
```

```
function bankers (proc : proctype; nump : integer) : boolean;



var
  numtapes : integer;
  lcv : integer;
  flag : boolean;



begin
  numtapes := MAXTAPES;
  for lcv := 1 to nump do
    begin
    numtapes := numtapes - proc[lcv].alloc;
    proc[lcv].maynotfin := true;
    proc[lcv].claim := proc[lcv].demand - proc[lcv].alloc;
    end;
  flag := true;
  while flag do
    begin
    flag := false;
    lcv := 1;
    while (lcv <= nump ) do
      begin
      if ( (proc[lcv].maynotfin) and
          (proc[lcv].claim <= numtapes))
        then begin
        proc[lcv].maynotfin := false;
```

```
            numtapes := numtapes + proc[lcv].alloc;
            flag := true;
            end;
          lcv := lcv + 1;
          end;
        end;
      if (numtapes = MAXTAPES)
        then bankers := false
      else
        bankers := true;
    end;
```

```
procedure trystart (var res : restype; var proc : proctype;
        pnum, nump, rnum : integer );

var
  dead : boolean;


begin
  proc[pnum].alloc := proc[pnum].alloc + rnum;
  proc[pnum].resources[TAPES].requested := false;
  dead := bankers (proc, nump);
  if (dead)
    then begin
    proc[pnum].alloc := proc[pnum].alloc - rnum;
    proc[pnum].resources[TAPES].requested := true;
    end
  else
    begin
    if (not proc[pnum].resources[TAPES].allocated) then
      enque(pnum,rnum,res[TAPES].toarc, res[TAPES].toarctail);
    deque ( pnum, res[TAPES].q, res[TAPES].tail );
    proc[pnum].req := 0;
    proc[pnum].resources[TAPES].allocated := true;
    res[TAPES].avail := res[TAPES].avail - rnum;
    end;
end;
```

```
procedure grantreq ( var res : restype; var proc : proctype;
                     nump : integer );



var
  t : intlist;



begin
  t := res[TAPES].q;
  while (t < > nil) do
    begin
    if (t¬.req < = res[TAPES].avail) then
      trystart (res, proc, t¬.pnum, nump, t¬.req);
    t:= t¬.next;
    end;
end;
```

```
procedure release (var res : restype; var proc : proctype;
        rnum, pnum, nump : integer );



begin
  readln;
  res[TAPES].avail := res[TAPES].avail + rnum;
  proc[pnum].alloc := proc[pnum].alloc - rnum;
  if (proc[pnum].alloc = 0) then
    begin
    deque (pnum, res[TAPES].toarc, res[TAPES].toarctail );
    proc[pnum].resources[TAPES].allocated := false;
    end;
  if res[TAPES].q < > nil
    then grantreq ( res, proc, nump );
end;
```

```
procedure request (var res : restype; var proc : proctype;
        rnum, pnum, nump : integer );

var
  dead : boolean;



begin
  readln;
  dead := true;
  if (res[TAPES].avail > = rnum) then
    begin
    proc[pnum].alloc := proc[pnum].alloc + rnum;
    proc[pnum].resources[TAPES].requested := false;
    dead := bankers (proc, nump);
    if (not dead)
      then begin
      if (not proc[pnum].resources[TAPES].allocated)
        then begin
        enque (pnum, rnum, res[TAPES].toarc,
          res[TAPES].toarctail);
        proc[pnum].resources[TAPES].allocated := true;
        end;
      proc[pnum].req := 0;
      res[TAPES].avail := res[TAPES].avail - rnum;
      end
    else
```

```
        proc[pnum].alloc := proc[pnum].alloc - rnum;
      end;
    if dead
      then begin
      proc[pnum].resources[TAPES].requested := true;
      proc[pnum].req := rnum;
      enque ( pnum, rnum, res[TAPES].q, res[TAPES].tail );
      end;
  end;
```

```
procedure skipblanks (var ch : char);

begin
   read ( ch );
    while (ch = BLANK) do
     read ( ch );
end;
```

```
procedure processline (var res : restype; var proc : proctype;
        nump : integer);

var
  ch : char;
  pnum : integer;
  com : char;
  rnum : integer;



begin
   skipblanks ( ch );
   if (ch = 'D') then
      begin
      readln;
      page ( output );
      writeln;
      display ( res );
      end
   else
      begin
      read ( ch );
      read ( pnum );
      skipblanks ( com );
      read ( ch );
      read ( rnum );
      case com of
```

```
        'Q' : request (res, proc, rnum, pnum, nump);
        'L' : release (res, proc, rnum, pnum, nump);
        'S' : sendmess (res, proc, rnum);
        'R' : receivemess (res, proc, rnum, pnum, nump);
        end;
        end;
end;
```

```
procedure simulate (var res : restype; var proc : proctype;
        nump : integer );



begin
    while not eof do
     processline ( res, proc, nump );
end;
```

```
procedure initprocess ( var res : restype; var proc : proctype;
        process : integer; var nump : integer );



var
  ch : char;
  demand : integer;
  resource : integer;



begin
   skipblanks ( ch );
   read ( ch );
   read ( demand );
   if not eoln then
      begin
      readln ( resource );
      new ( res[resource].toarc );
      res[resource].toarc¬.pnum := process;
      res[resource].toarc¬.next := nil;
      proc[process].resources[resource].producer := true;
      proc[process].resources[resource].requested := false;;
      proc[process].resources[resource].allocated := false;
      end;
   proc[process].demand := demand;
   proc[process].alloc := 0;
   proc[process].req := 0;
   nump := nump + 1;
```

```
end;
```

```
procedure initsys (var res : restype; var proc : proctype;
        var nump : integer );

var
  processnum : integer;
  ch : char;



begin
  processnum := -1;
  res[TAPES].avail := MAXTAPES;
  skipblanks ( ch );
  read ( ch );
  read ( processnum );
  while (processnum < > 0) do
    begin
    initprocess ( res, proc, processnum, nump );
    skipblanks ( ch );
    read ( ch );
    read ( processnum );
    end;
  readln;
end;
```

```
begin
  initsys ( res, proc, nump );
  simulate ( res, proc, nump );
end.
```

Appendix D. Operating Systems Assignment

CS5661

Spring, 1987

Dr. Sallie Henry

Due Dates

Design: April 23, 1987

Code, Revised Design, and Output: April 30, 1987

Points: 50

Motivation

The motivation for this assignment is twofold. First, you will use a modern design language to aide in the architectural and detailed design phases of the software lifecycle. Design languages are tools created to make the design process easier, and more efficient. This in turn helps to expedite the implementation and testing phases of the software lifecycle, and to help us to produce more reliable software. Second, you will simulate management of consumeable and reusable operating system resources to prevent and detect deadlock.

Constraints

Our system will use the following constraints:

1. 10 tape drives (our reusable resource!)

2. 5 message areas (our consumeable resource!)

3. Any number of active processes

4. A FIFO queue associated with each appropriate resource

5. Bankers algorithm for automatic deadlock prevention of the reusable resource

6. A deadlock detection algorithm for the message areas.

note:

a single queue is needed for management of the tape drives, because a process doesn't care which tapes it gets, but, each message area will require its' own queue. When resources become available, the first member of a queue is checked to see if there are enough available. If so, the resources are granted. If not, continue checking the queue and grant any of the resources that you can.

When deadlock is detected simply flush the inventory and queues of each of the 5 message areas.

Input

At the beginning of the input there will be a series of initialization lines that give the maximum demand for tapes and the number of the consumeable resource for which the process is a producer, if it is one. The end of the initialization section will be flagged by reading in a process number of 0.

The initialization cards will have the following format:

1. Producer process initialization

process	max. demand	produces
P = 3	D = 2	1

2. Non-procducer process initialization

P = 2 D = 5

Input consists of the following types and formats:

1. Tape request

process	number of tapes
P = 1	Q = 4

2. Tape release

process	number of tapes
P = 3	L = 1

3. Send a message

process	resource number	number of messages
P = 5	S = 3	4

4. Receive a message

process	resource number	number of messages
P = 2	R = 1	2

5. A display of all of the jobs in the system

   D

All input will be syntactically correct, and there will also be at least one blank after every number.

Output

Output should consist of the following:

1. Whenever a display is called for in the input file, the contents of each queue should be displayed

2. Whenever deadlock is detected, output a message to that effect, *and* display the contents of each queue.

Grading

Your design will be an integral part of your grade (so don't just throw something together!). It will be graded on both quality and completeness. The design you hand in with your working system code should reflect any changes that were made during the implementation process. If you've done a good job of designing these changes should be minimal at best.

Initial Design : 50%

Revised Design and Code : 25%

Results: 25%

There will be a help session on the design languages at 4:00 on Monday the 20th in McB 328.

Appendix E. GPL Regression Line and Confidence Interval Plots

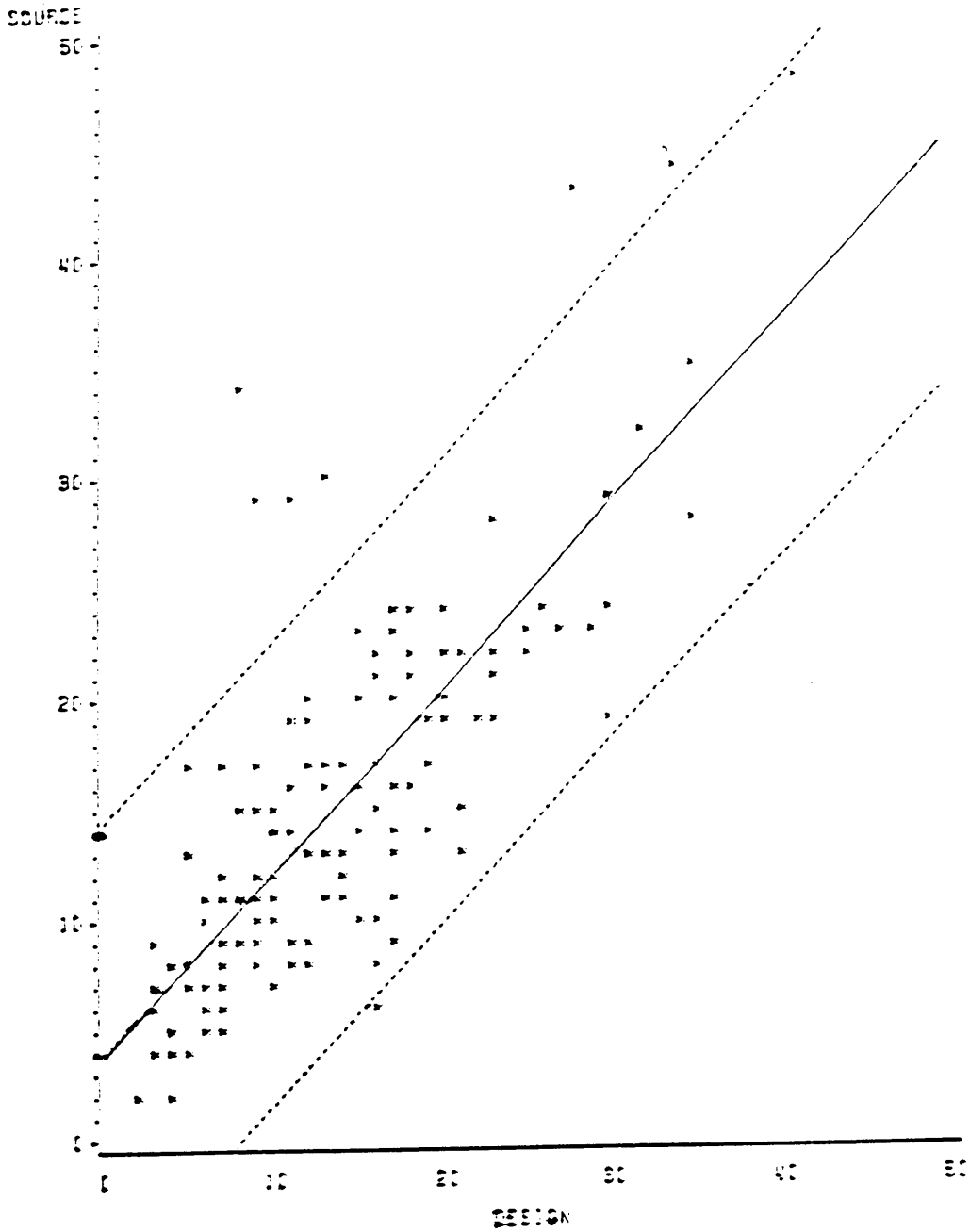


Figure 8. GPL Length Regression and 95% Confidence Lines

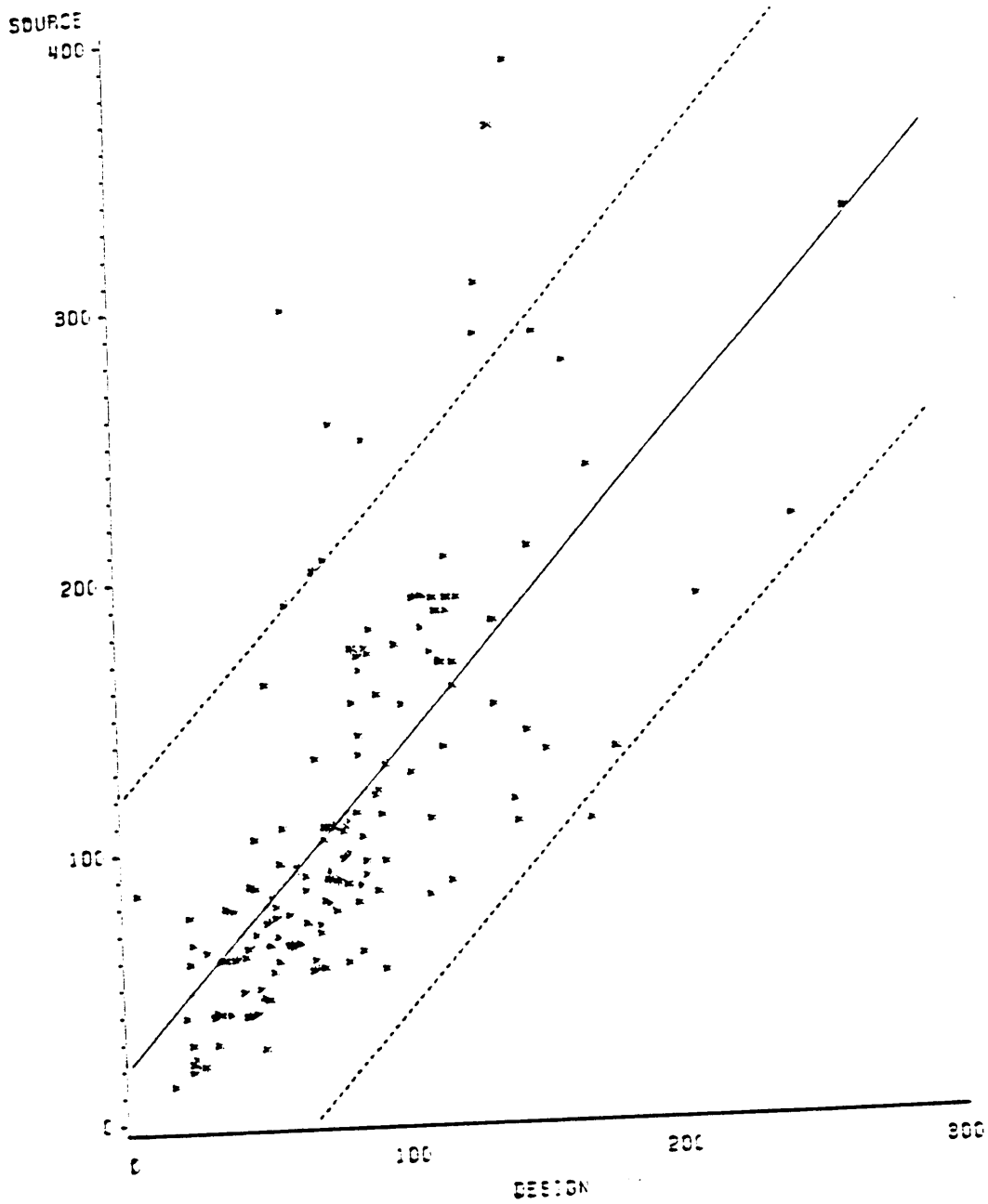


Figure 9. GPL N Regression and 95% Confidence Lines

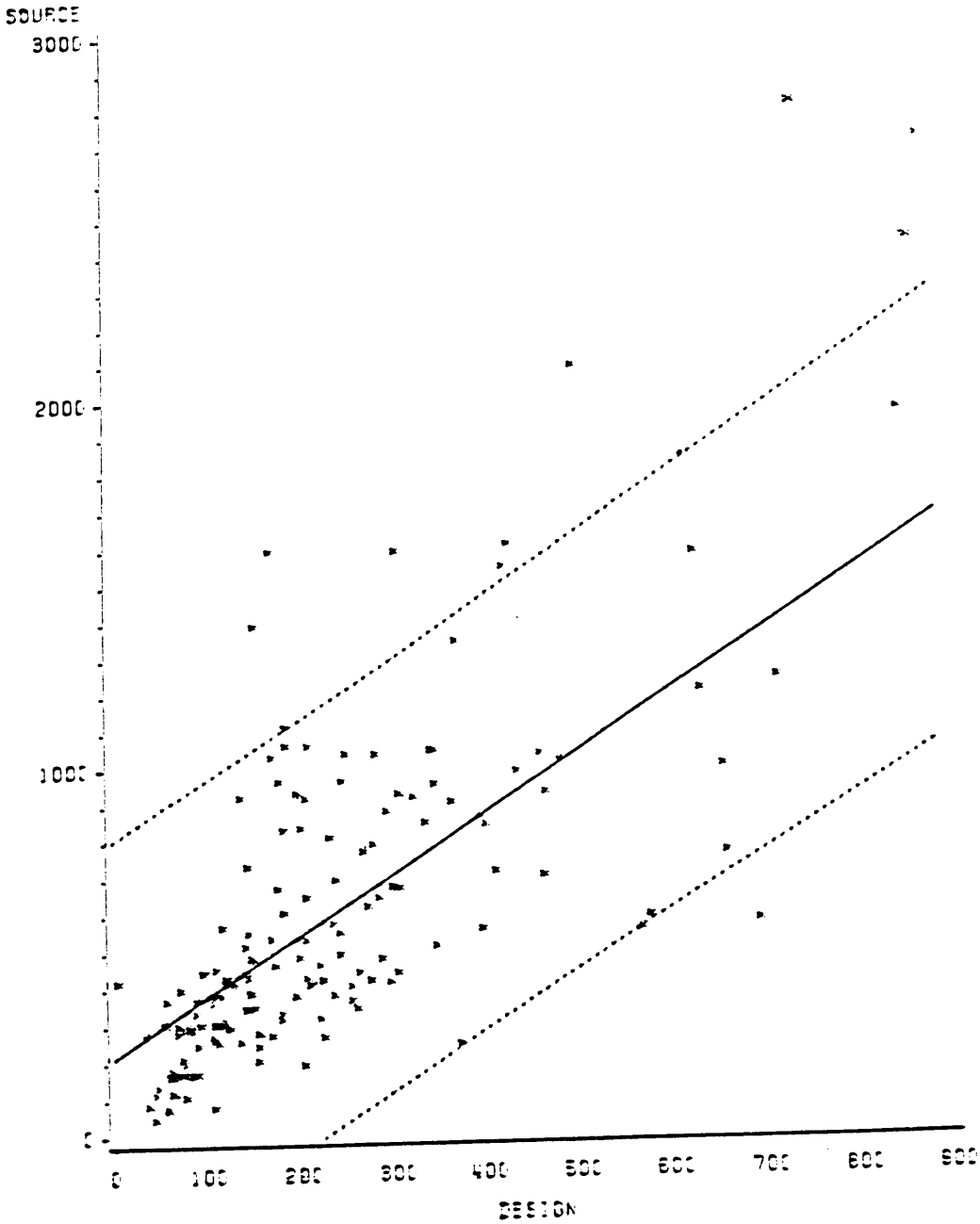


Figure 10. GPL Volume Regression and 95% Confidence Lines

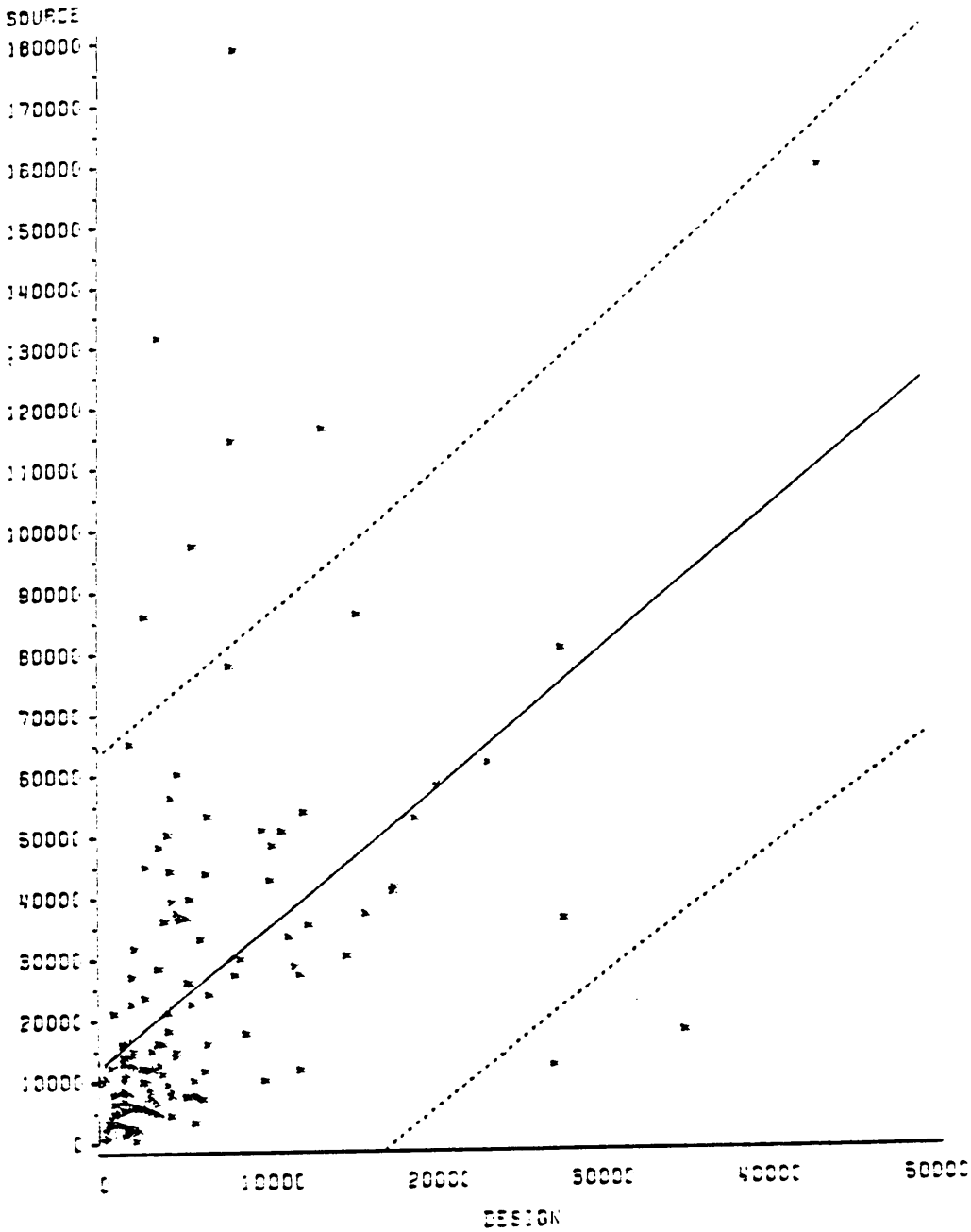


Figure 11. GPL Effort Regression and 95% Confidence Lines

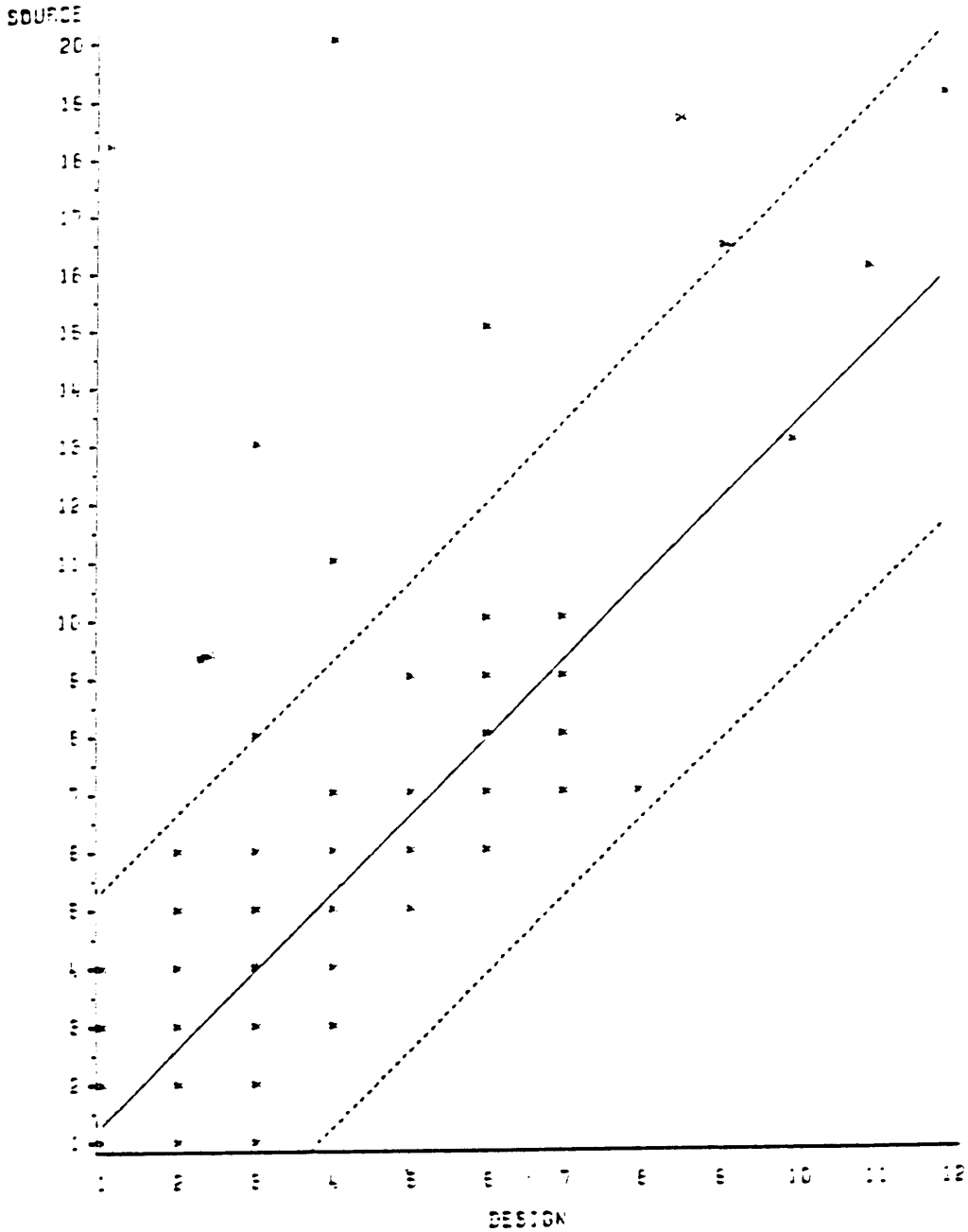


Figure 12. GPL Cyclomatic Regression and 95% Confidence Lines

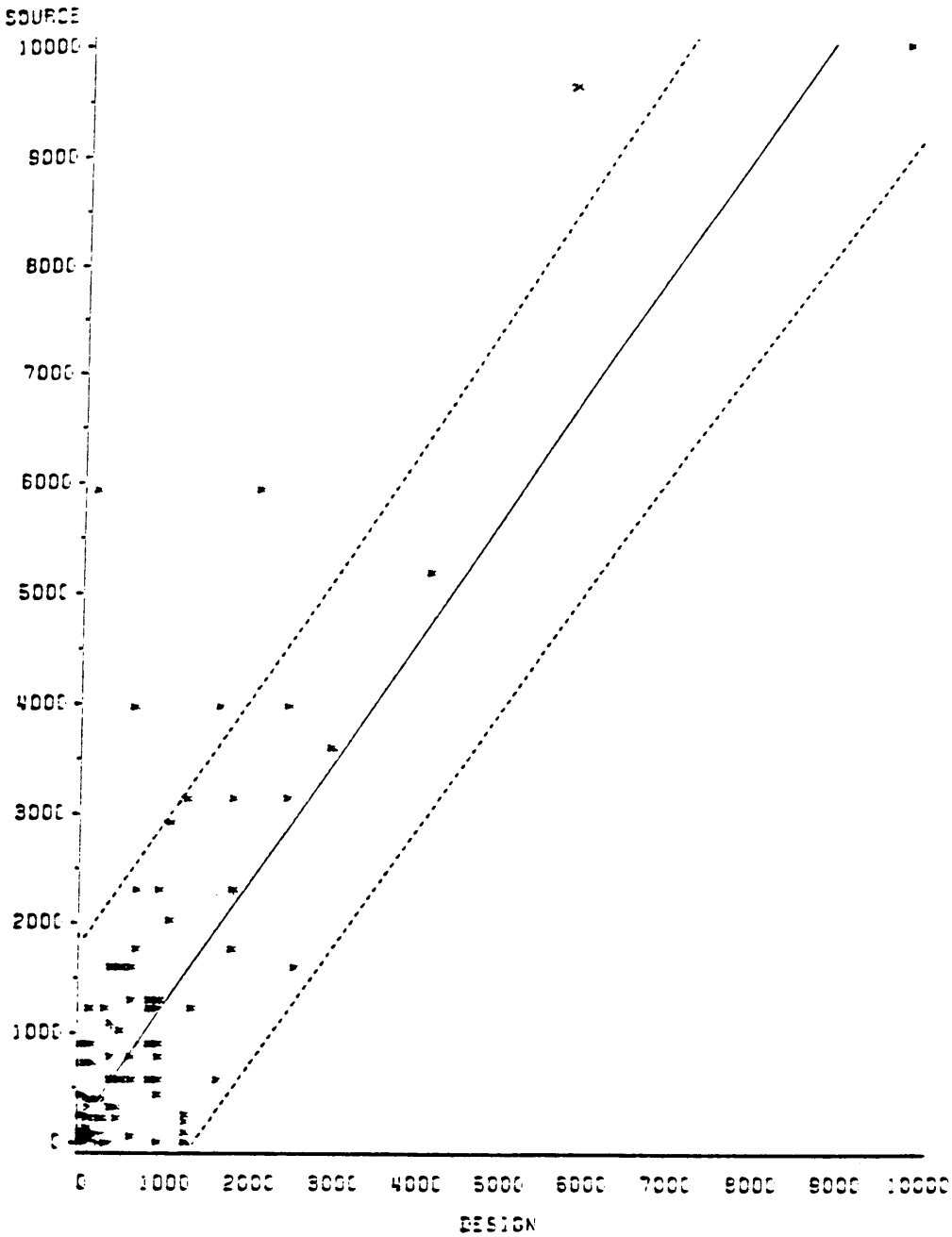


Figure 13. GPL Information Flow Regression and 95% Confidence Lines

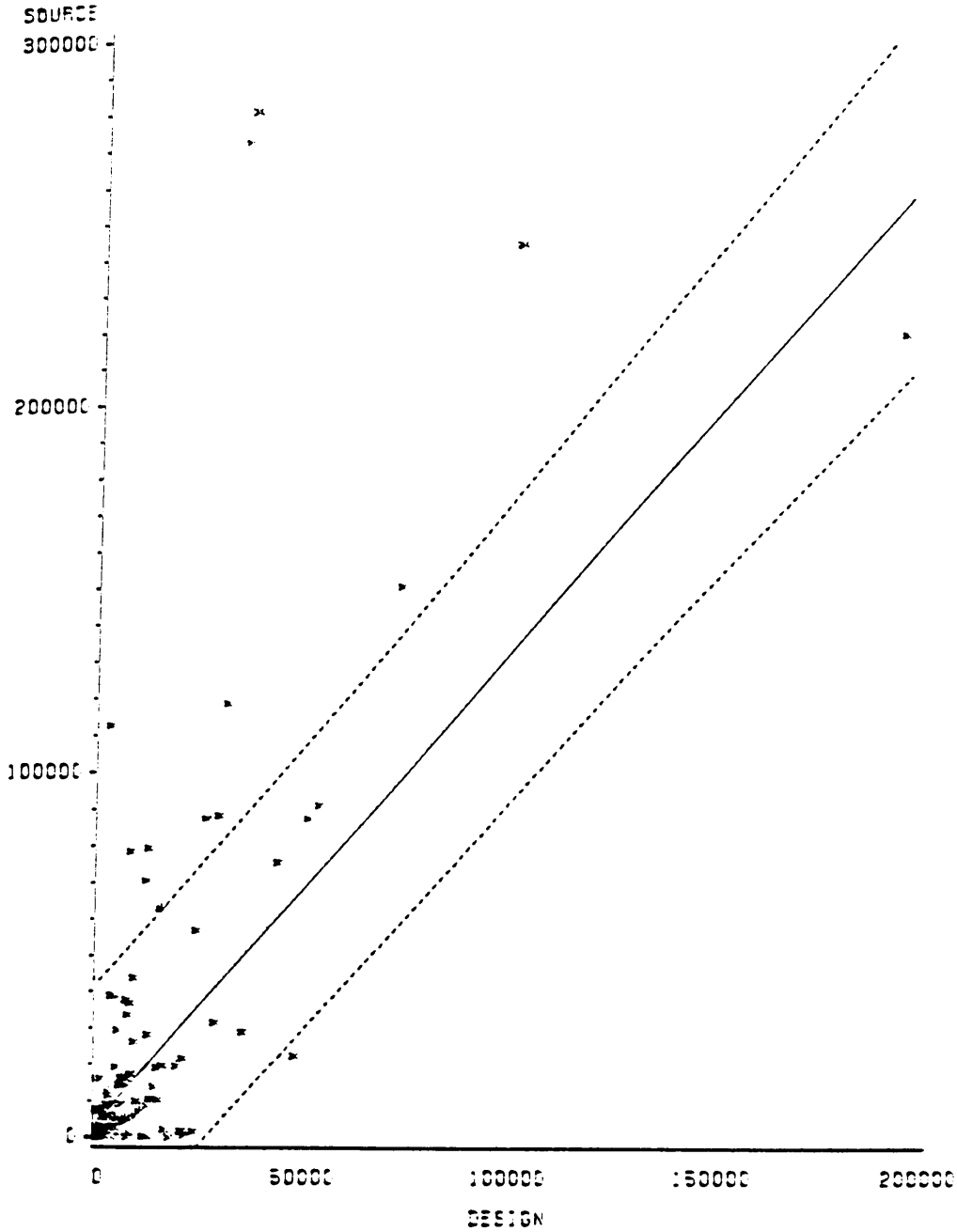


Figure 14. GPL Information Flow with Length Regression and 95% Confidence Lines

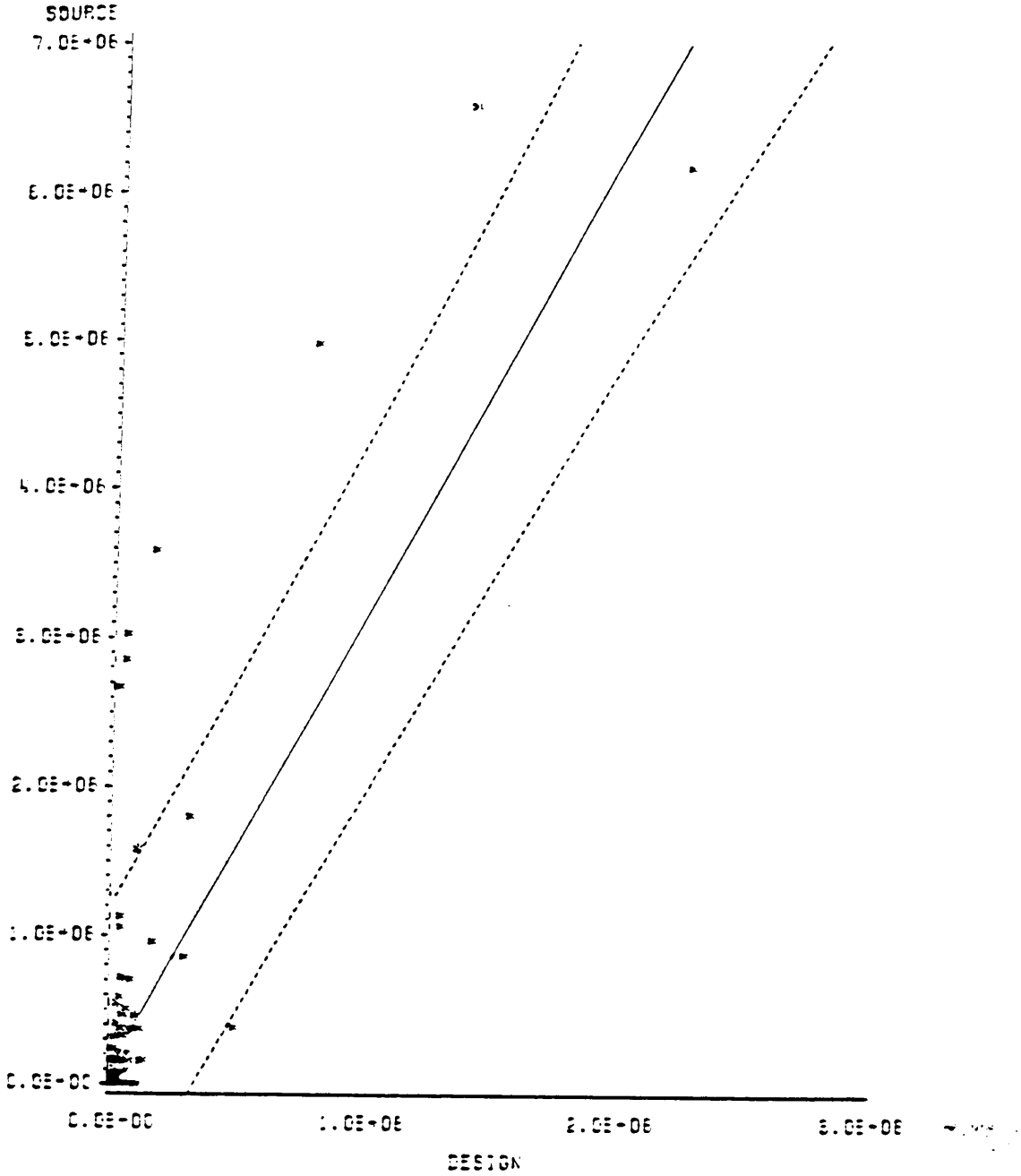


Figure 15. GPL Information Flow with Effort Regression and 95% Confidence Lines

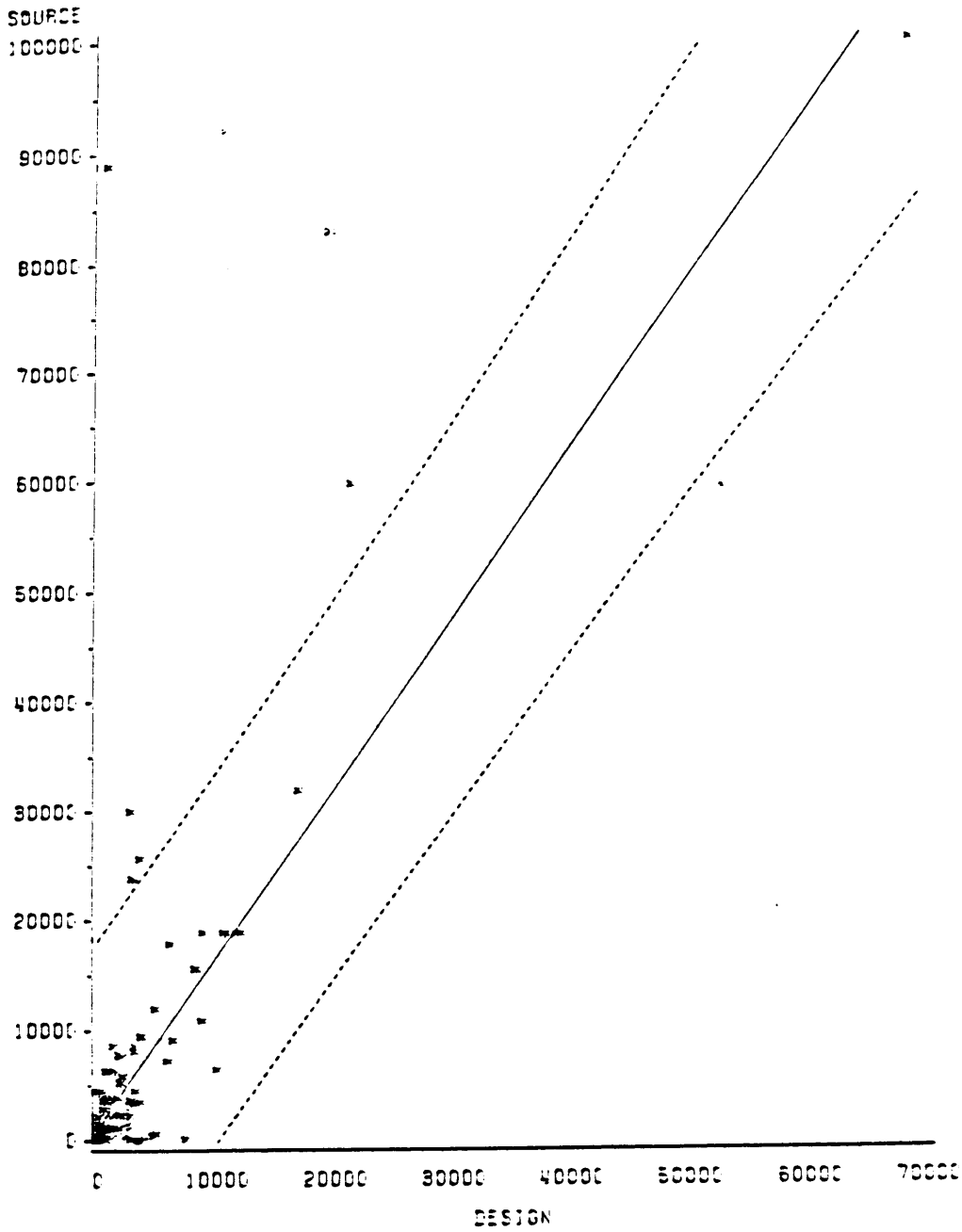


Figure 16. GPL Information Flow with Cyclomatic Regression and 95% Confidence Lines

Appendix F. ADLIF Regression Line and Confidence Interval Plots

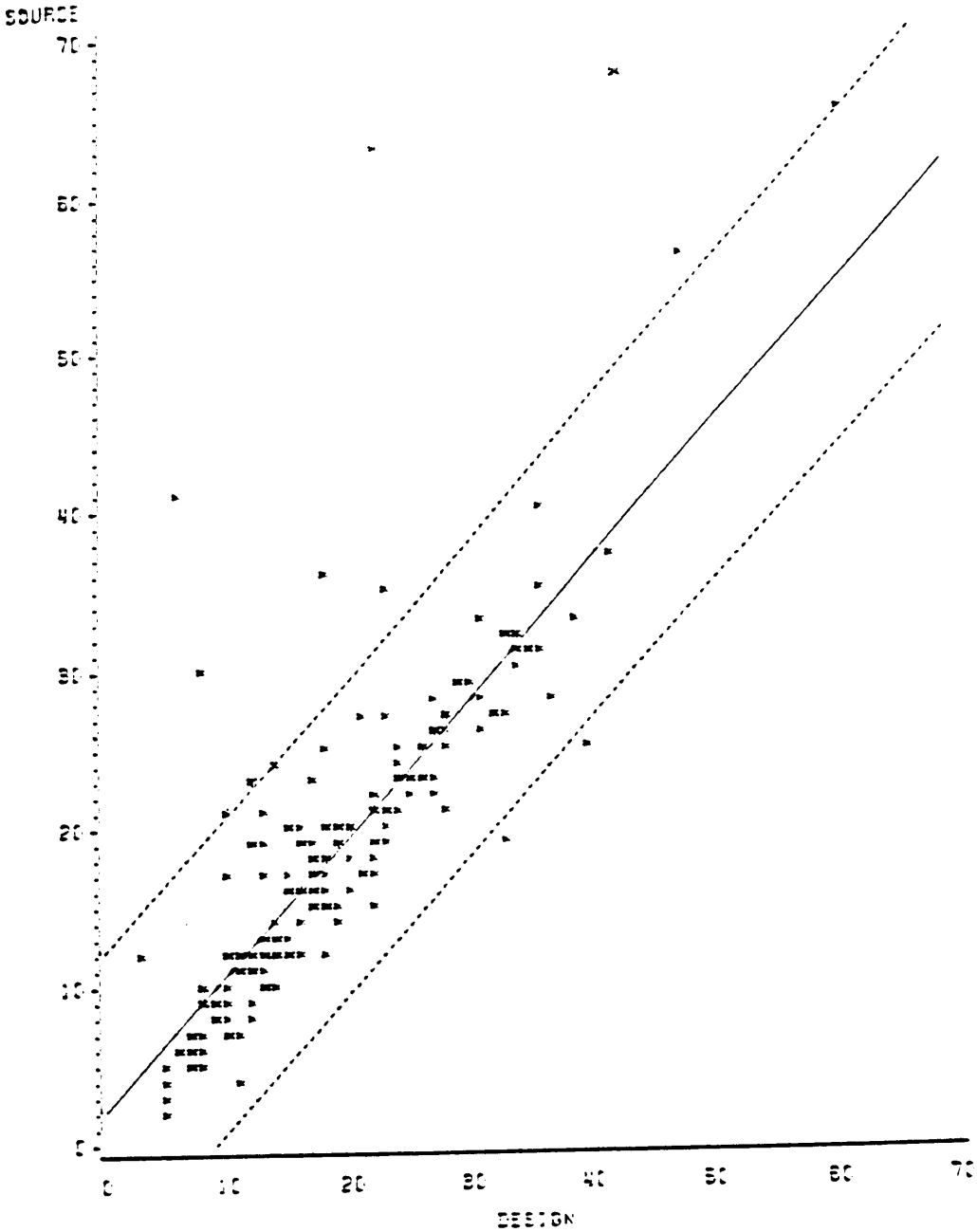


Figure 17. ADLIF Length Regression and 95% Confidence Lines

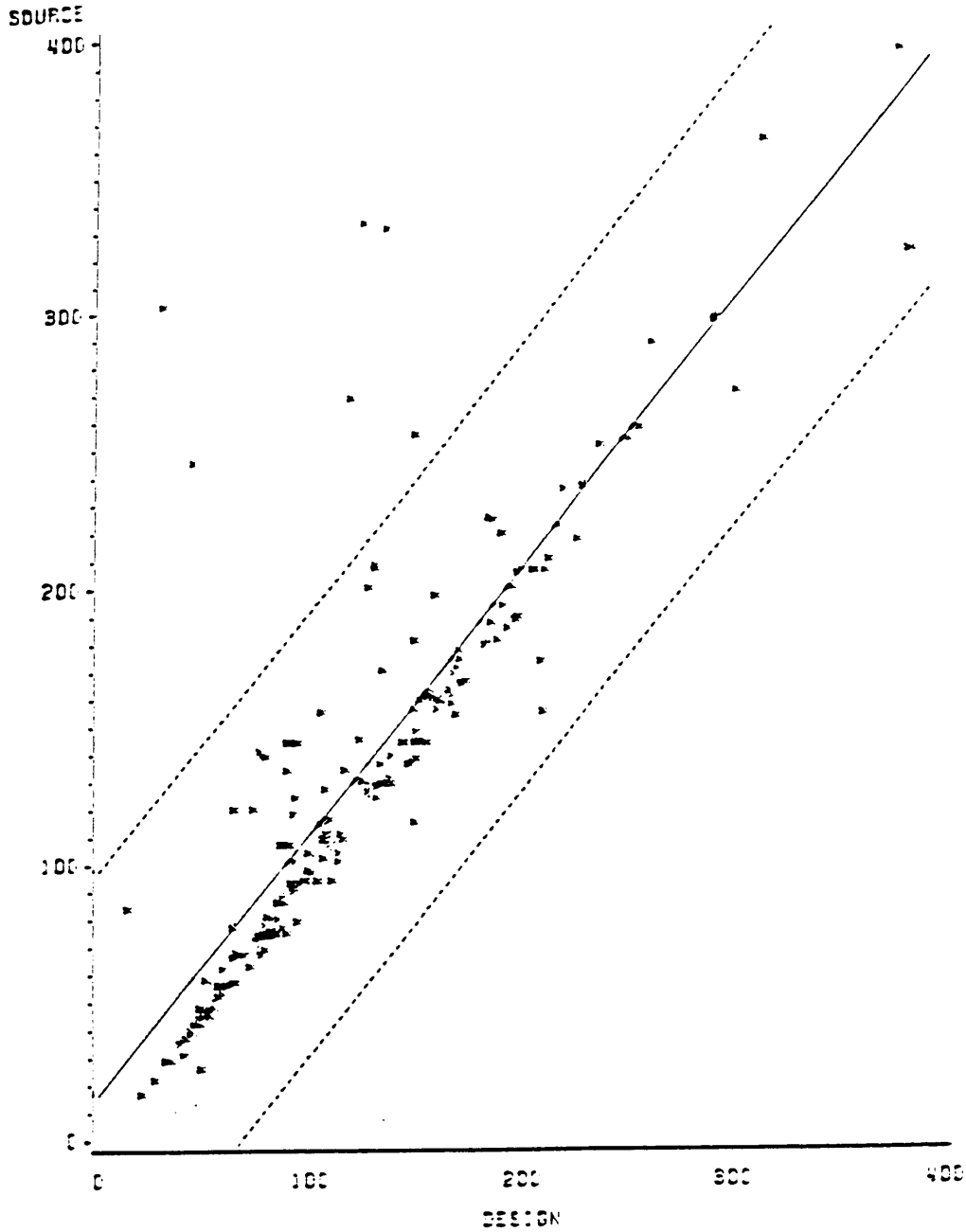


Figure 18. ADLIF N Regression and 95% Confidence Lines

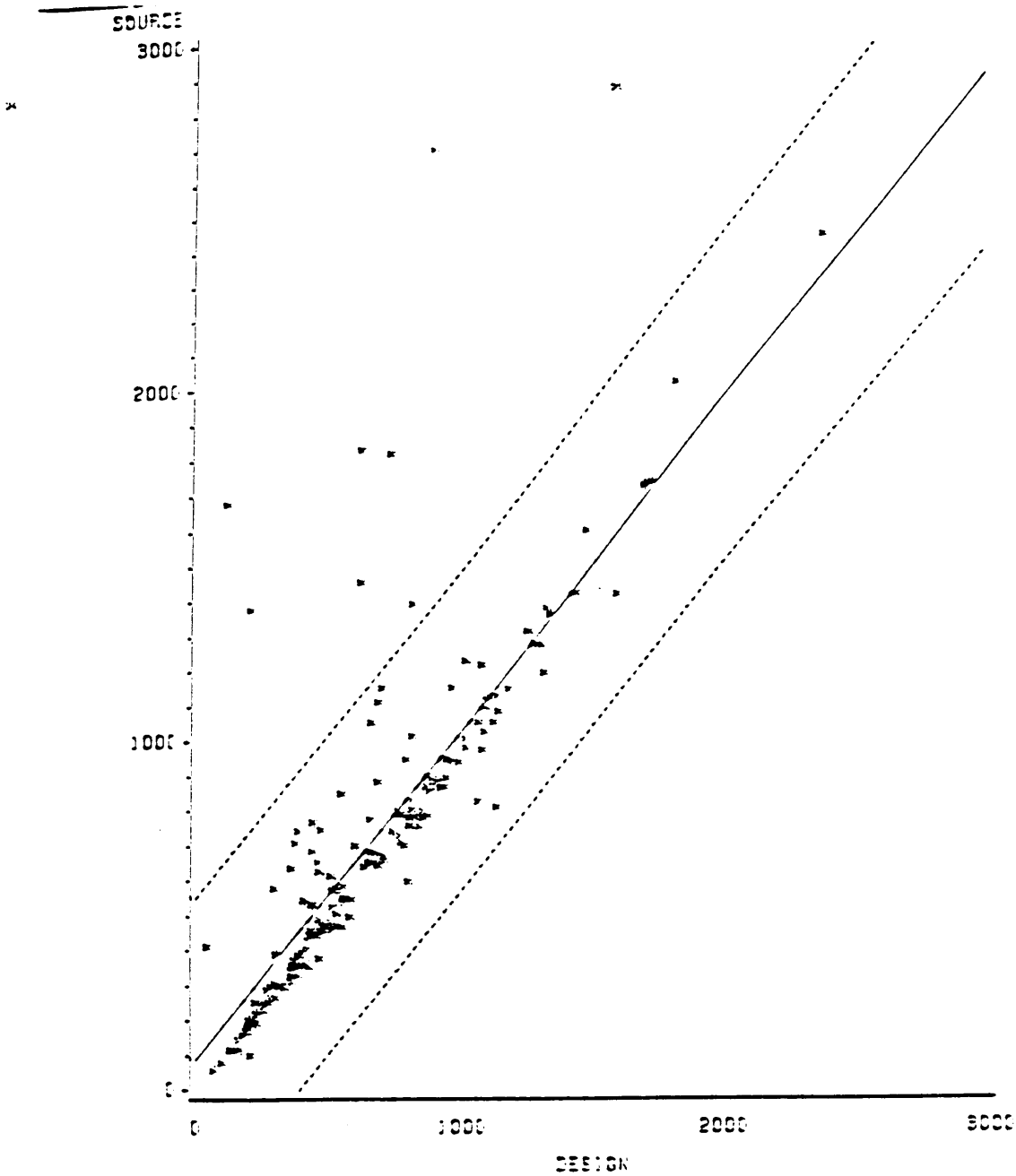


Figure 19. ADLIF Volume Regression and 95% Confidence Lines

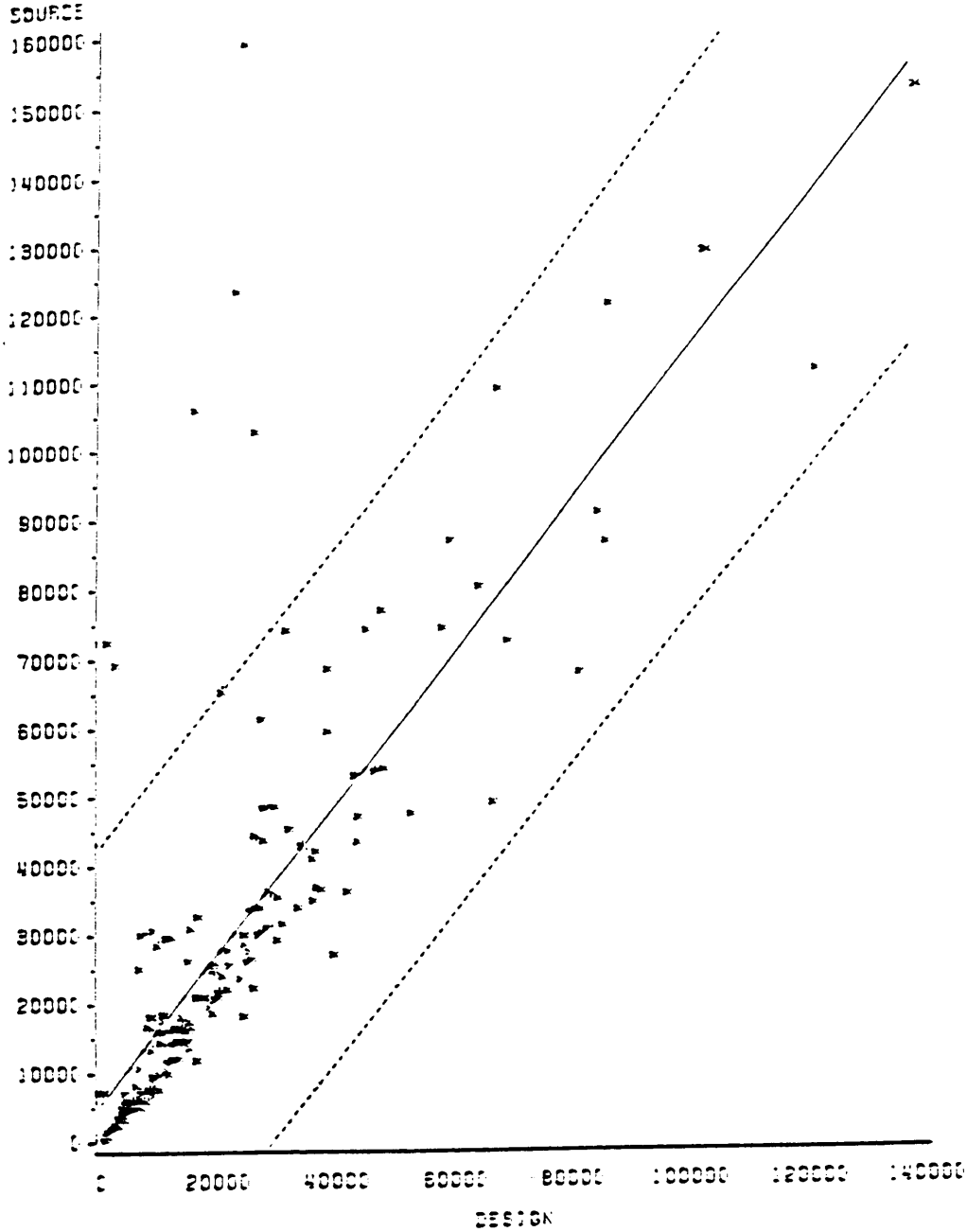


Figure 20. ADLIF Effort Regression and 95% Confidence Lines

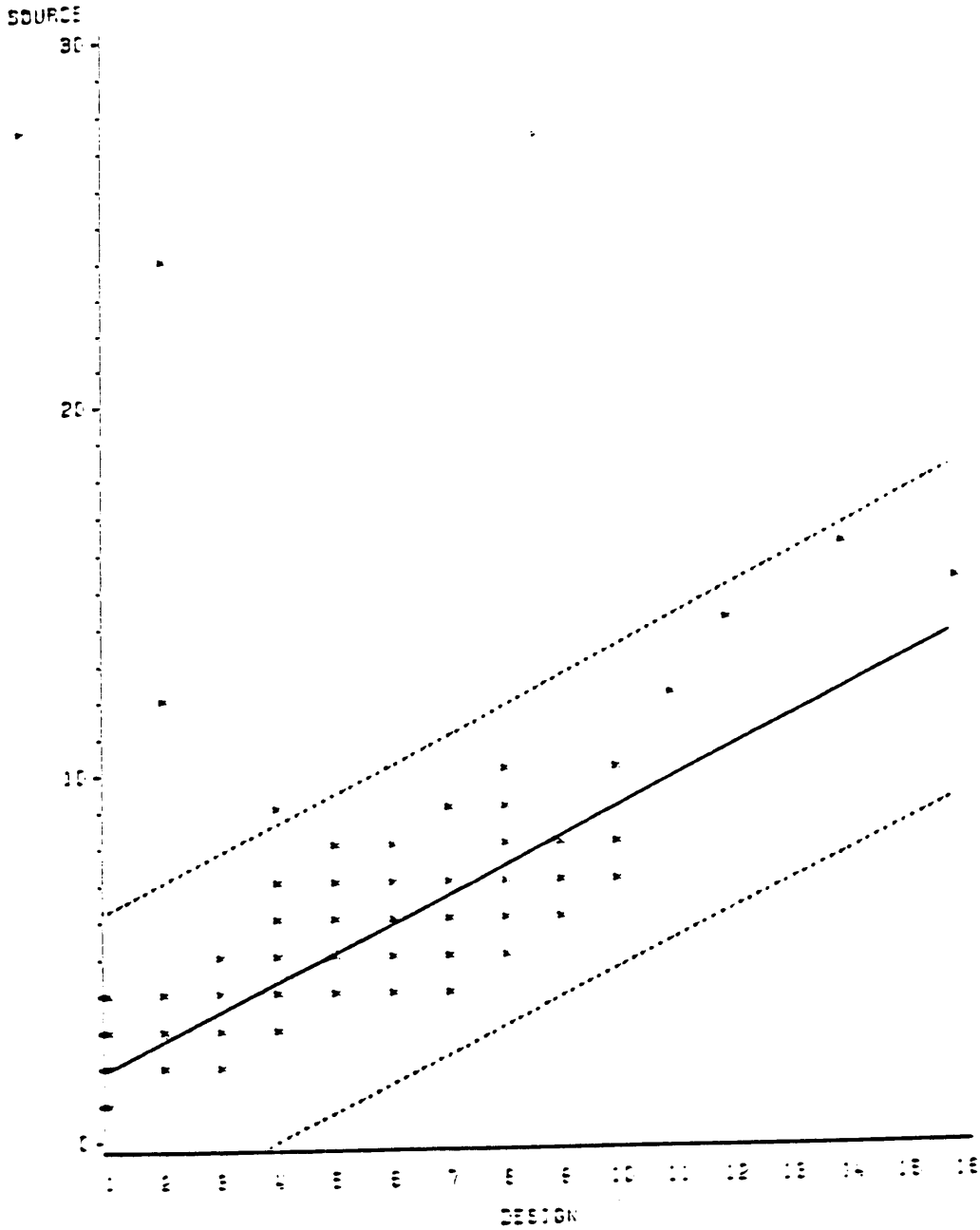


Figure 21. ADLIF Cyclomatic Regression and 95% Confidence Lines

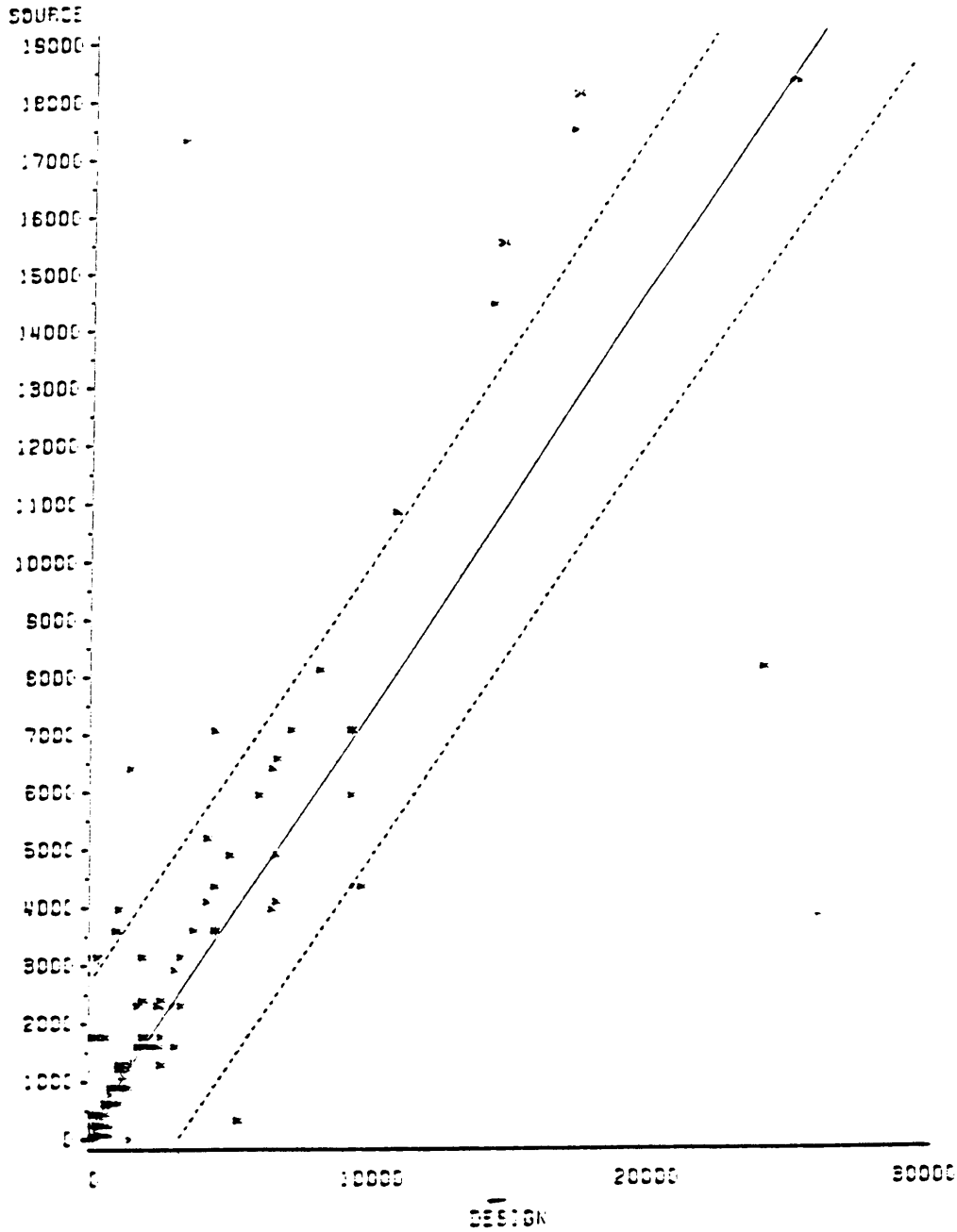


Figure 22. ADLIF Information Flow Regression and 95% Confidence Lines

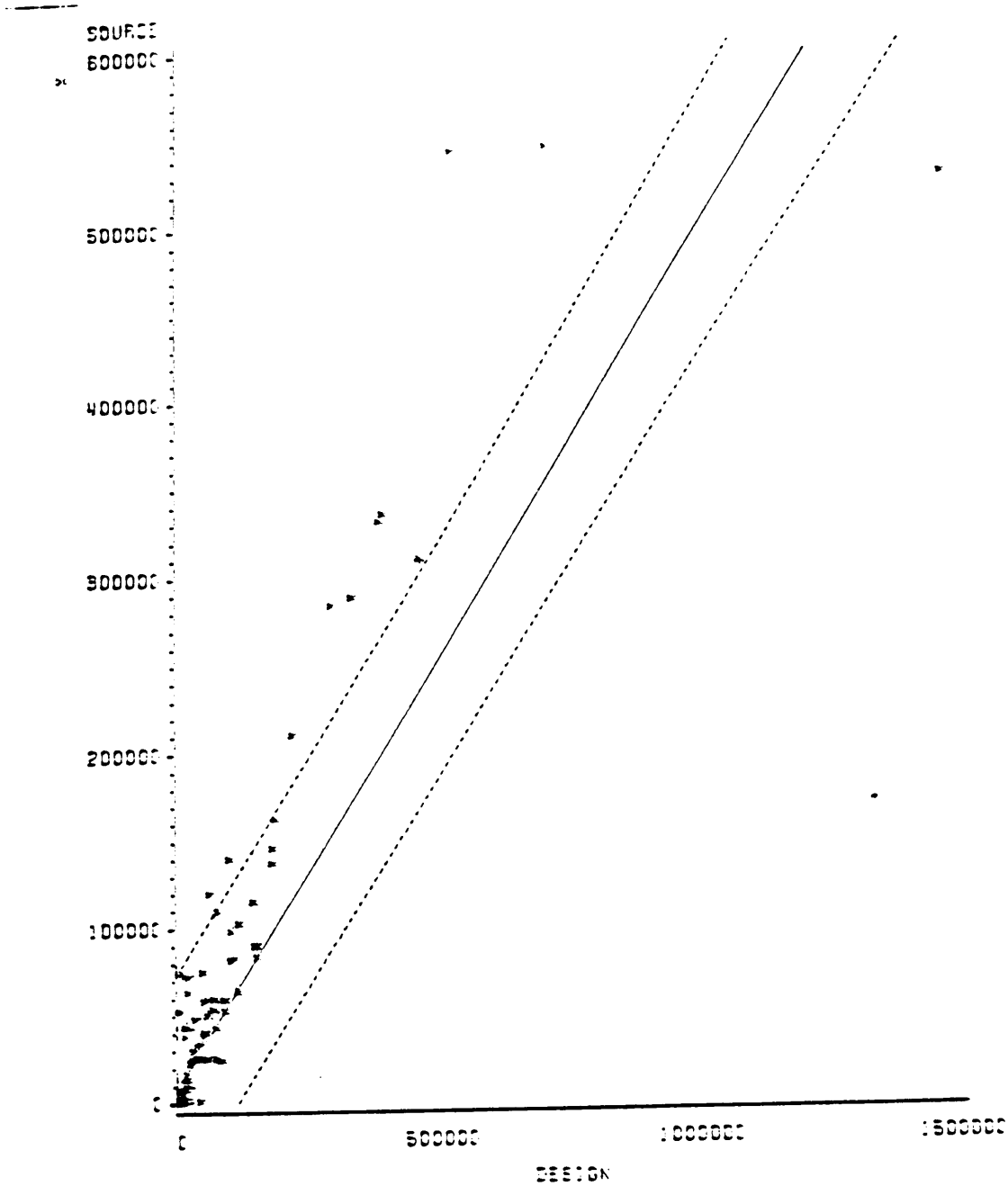


Figure 23. ADLIF Information Flow with Length Regression and 95% Confidence Lines

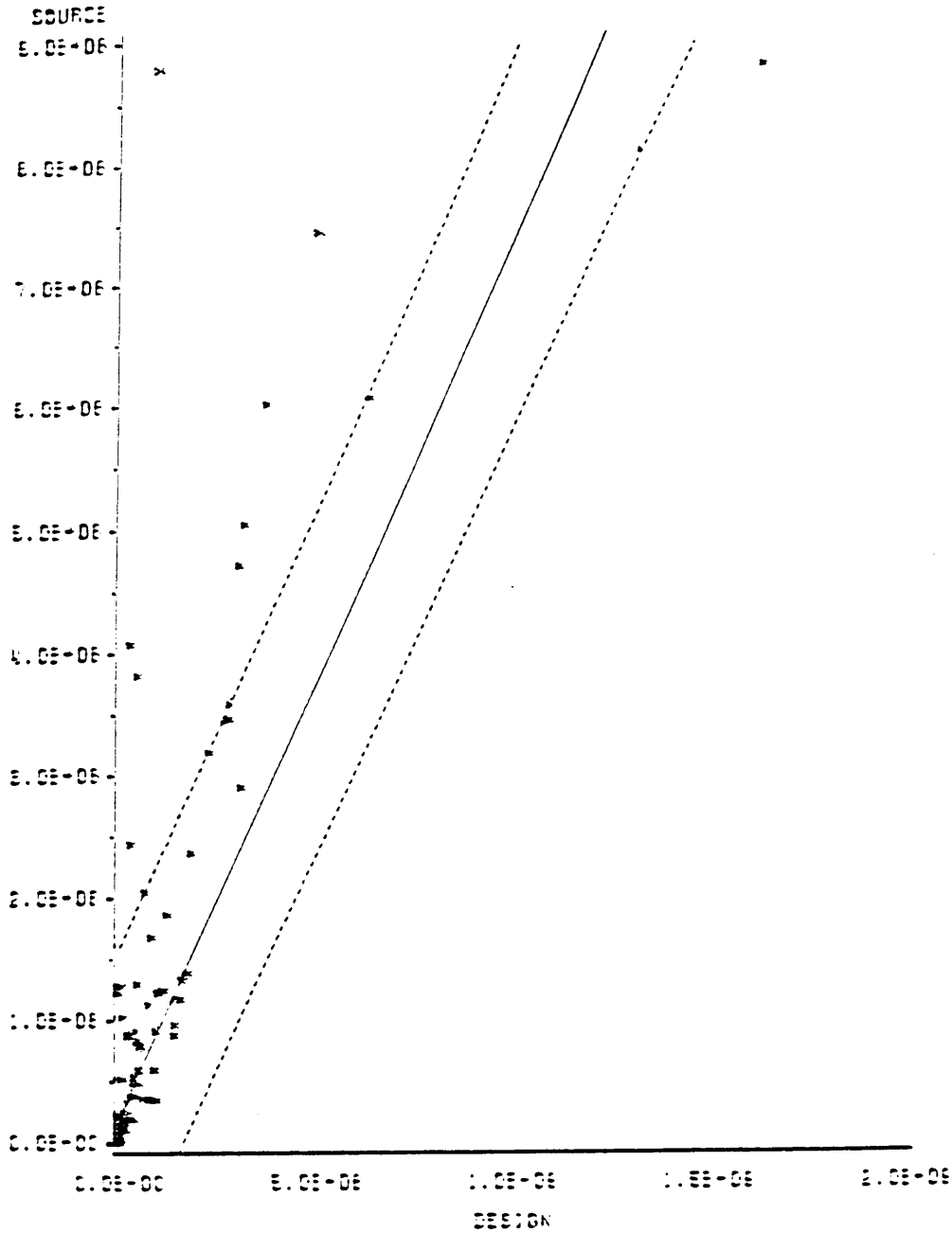


Figure 24. ADLIF Information Flow with Effort Regression and 95% Confidence Lines

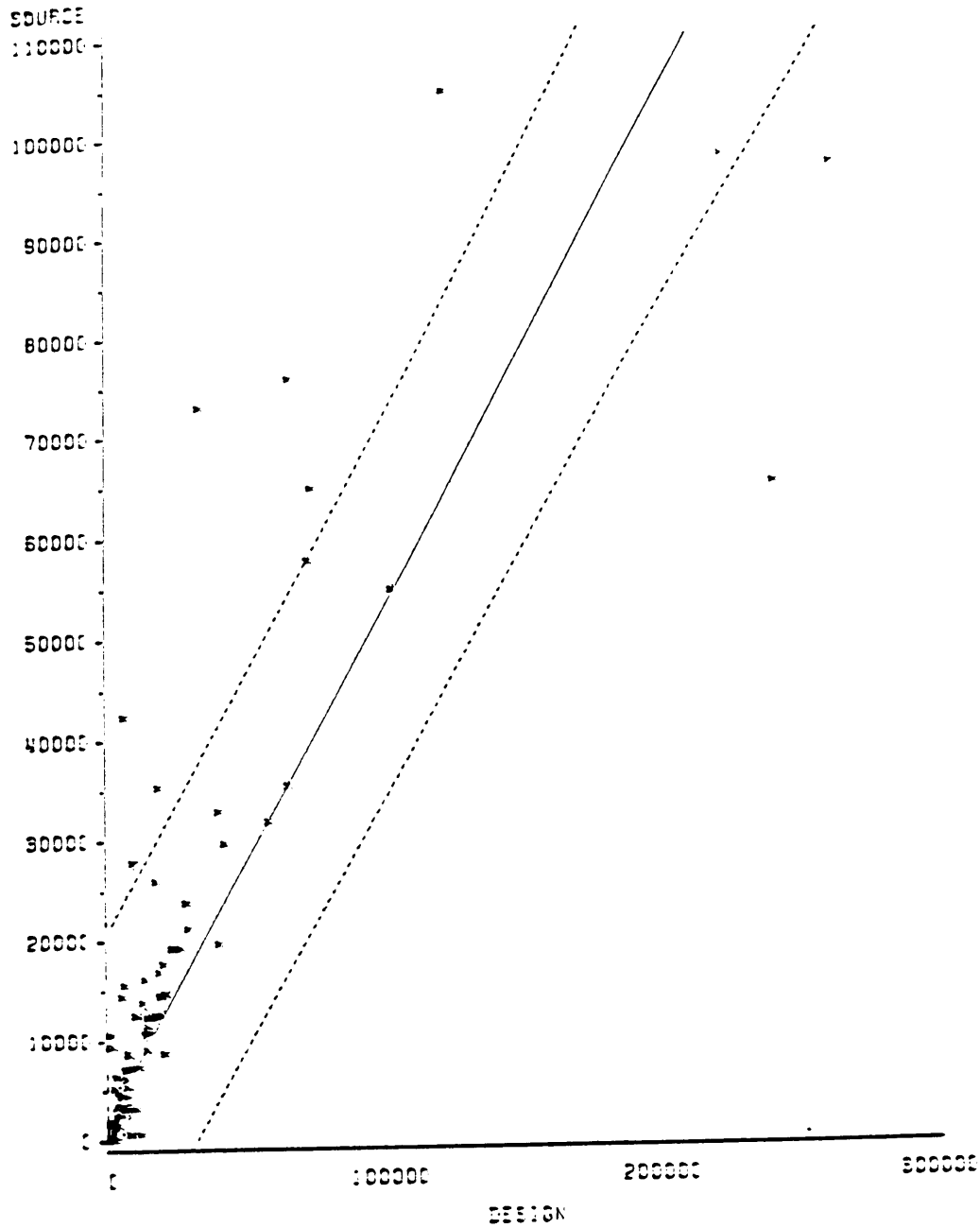


Figure 25. ADLIF Information Flow with Cyclomatic Regression and 95% Confidence Lines

Appendix G. Software Engineering Assignment

CS4980 Final Exam

Your assignment, should you decide to accept it (well, actually you must accept it), is to design and implement the "robots" game. The following is what the board to the game must look like:

```
| <----------------60 spaces------------------ > |

+-------------------------------------------+        MOVES
|                                           |
|                                           |    y   u   i
|                                           |     \ | /
|                                           |    h - j - k
|                                           |     / | \
|                                           |    n   m   ,
|                                           |
|                                           |
|                                           |    COMMANDS
|                                           |
|                                           | t - teleport
|                                           | s - last stand
|                                           | q - quit
| L - redraw                                |
|                                           |
|                                           |
|                                           |    KEY
|                                           |
```

```
|                                    | I - you
|                                    | = - robot
|                                    | @ - junk heap
|                                    |
+----------------------------------------+      SCORE

                                        0
```

The object of the "robots" game is to get all the robots, who are trying to get you, to run into each other, or into junk heaps. When two robots collide they both die, and a junk heap is created. You can move around the playing area using the keys shown in the MOVES area of the display. You may not move outside the playing area, any attempt to do so should be counted as a move. For example, if you are on the lefthand edge of the playing area and you attempt to do an "n" you should leave yourself on the lefthand edge but move yourself down one row. If you tried to do an "h" you should leave yourself in the same place, etc.. Every time that you make a move, each robot also makes a move.

Note:

Striking any other keys than those of the commands or moves does nothing, and does not count as a move.

Commands

- The "teleport" command randomly generates a new position for you in the playing area. You can teleport at anytime. Teleporting counts as a move so, after your new position has been determined and you have moved there, the robots all get a move.

- The "last stand" command has the same effect as doing "j"s, i.e. don't move anywhere, until you die or all the robots die.

- The "quit" command quits the game.

- The "redraw" command simply redraws the screen.

The Play

Initially the playing area has ten robots and you on it. You get the first move, then all the robots get a move, then you, and so on. If you succeed in killing all of the robots you have won the round. The next round will start with twenty new robots and none of the junk heaps created in the previous round, then thirty new robots and none of the old junk heaps, then forty, etc.. Play is ended when you die.

Note:

> Don't forget that when you are generating positions for the robots it is possible that you will generate two robots to be on the same position. This is called a collision, and your program must handle collisions by providing a different position for the colliding robot.

Death

Death is signified by printing "*MUNCH*" at the position on the screen where you died, and is achieved in one of three ways:

1. A robot moves onto the space where you are

2. You move onto a junk heap

3. Teleporting onto a place where there is already a robot

Scoring

Your score is printed in the lower right hand corner of the screen (see the diagram above) and is updated continuously. Every robot that you kill is worth the same number of points that there were robots on the screen at the beginning of the round.

The Robots

The robots are not too intelligent, and their moves are very simple. If you are on the same row or in the same column, the robots simply take one step in your direction. Otherwise, the robots take one diagonal step in your direction. The robots are not intelligent enough to look and see if some other robot is going to move to the same place that they are, nor do they look to see if there is a junk heap already on the spot they are going to move to.

The Assignment

You are to design, in ADLIF or GPL, and implement, in PASCAL, the "robots" game.

The ADLIF group will be using the VAX35 account:

CS49800F

password: adliff

The GPL group will be using Zonker, under Roger's supervision, for their designs and the VAX35 account

CS498015

password: graphic

Th VAX35 accounts have a sub-directory, by last names, for each of the persons who will be using that account. Please work only under your sub-directory.

The vita has been removed from the scanned document