Comparison of a Graphical and a Textual Design Language Using Software Quality Metrics

Sallie Henry
Roger Goff

TR 88-20

Comparison of a Graphical and a Textual Design Language Using Software Quality Metrics

by

Sallie Henry, Ph.D.
and
Roger Goff

Computer Science Department
Virginia Tech
Blacksburg, VA 24061

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. A comparison of a graphical and a textual design language is presented in an effort to support research findings that the human brain works more effectively in images than in text.

I. Introduction

In general, the software life cycle of requirements definition, program design, implementation, testing, and finally, maintenance. The portion of the cycle that is of interest to this research is that of design and implementation with the inclusion of software quality metrics. Figure 1 contains a diagram of this part of the software life cycle using complexity metrics. First, a design is created and implemented in software. At that point, software quality metrics are generated for the source code. If necessary, as indicated by the metrics, the cycle returns to the design phase. Ideally, the software life cycle can be "reduced" to that in Figure 2 where the metrics are generated during the design phase, before code implementation. This modified cycle will eliminate the generation of undesirable source code, since it is possible to use the metrics, exactly as before, only earlier. The goal of this study is to indicate the plausibility of using the "reduced" cycle to increase the efficiency of the software development process by implementing metric analysis as early as possible. Applying metrics during design was suggested by [1].

The goal of shortening the loop in the life cycle is highly dependent on the ability to perform the metrical measures on the design, along with the need for evidence that the metric values produced from the design reflect the quality of the resultant source code. To facilitate this ability, a software metric analyzer is provided that takes as input either the design or the source code and produces, as output, a number of complexity metric values.

This research is a quantitative comparison of a textual design tool to a graphical design tool. Software quality metrics are used to measure both designs and resultant source code. A brief description of the design tools used in this study follows.

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 a PDL are 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.

The textual design language, or PDL, used in this study has some Pascal-like and Ada-like constructs, but it is sufficient to say this design tool is a general purpose PDL. For a complete definition of the PDL, the interested reader is referred to [2].

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. 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.

GPL is the graphical programming language of the Dialogue Management System, DMS, being developed at Virginia Tech [3] and is the graphical design tool used in this research. 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 basic building block for a GPL computational design is a Supervisory Cell which contains a supervisor and a Supervised Flow Diagram (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 3 presents the symbols in GPL used for computational design. The syntax and semantics of each symbol and the information associated with each supervisor is discussed.

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. 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 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.

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.

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. Thus, there are no functions that return values in GPL.

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.

Databoxes

GPL databoxes are used to specify the actual input and output parameters to a subroutine call. They are sometimes called binding boxes since they contain 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.

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.

Functions

The function symbol represents a call to a subroutine that contains no dialogue. A GPL function is different from 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.

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 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.

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.

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.

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 which are needed for readability purposes, and second is the inability to define libraries of routines for the purpose of reusability.

The interpretation of graphical symbols to the more standard token counts used by a number of metrics is mostly intuitive. However, the exact translation is available for the interested reader in [4].

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.

Section 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. The results of measuring the graphical language are presented in section three and a comparison of the graphical language and a textual design language is given in section four. Finally, section five recounts our conclusions.

II. Metrics and the Analysis Tool

A brief description of the metrics used in this research is presented in this section. For the interested reader desiring more detailed information on the metrics please see the references.

Code Metrics

Many code metrics have been proposed in the recent past. An effort has been made to limit this discussion to a few of the more popular ones that are typical of this type of measure. They include lines of code, parts of Halstead's Software Science, and McCabe's Cyclomatic Complexity. Each of these metrics is widely used and has been extensively validated [5] [6] [2].

Lines Of Code

The most familiar software measure is the count of the lines of code with a unit of *LOC*. or, for large programs, *KLOC* (thousands of lines of code). Unfortunately, there is no consensus on exactly what constitutes a line of code. Most researchers agree that a blank line should not be counted but cannot agree on comments, declarations, null statements such as the Pascal “begin,”

etc. Another problem arises in free format languages which allow multiple statements on one textual line or one executable statement spread over more than one line of text.

For this study, the definition used is the following: A line of code is counted as the line or lines between semicolons, where intrinsic semicolons are assumed at both the beginning and the end of the source file. This specifically includes all lines containing executable and non-executable statements, program headers, and declarations.

Halstead's Software Science

A natural weighting scheme used by Halstead in his family of metrics (commonly called Software Science indicators [7]) is a count of the number of "tokens," which are units distinguishable by a compiler. All of Halstead's metrics are based on the following definitions:

n_1 = the number of unique operands.

n_2 = the number of unique operators.

N_1 = the total number of operands.

N_2 = the total number of operators.

Three of the software science metrics, *N*, *V*, and *E*, are used in this research.

The metric *N* is simply a count of the total number of tokens expressed as the number of operands plus the number of operators, i.e., $N = N_1 + N_2$.

V represents the number of bits required to store the program in memory. Given *n* as the number of unique operators plus the number of unique operands, i.e., $n = n_1 + n_2$, then $\log_2 (n)$ is the number of bits needed to encode every token in the program. Therefore, the number of bits necessary to store the entire program is:

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

The final Halstead metric examined is effort (*E*). The effort metric, which is used to indicate the

effort of understanding, is dependent on the volume (*V*) and the difficulty (*D*). The difficulty is estimated as:

$$D = n1/2 \times N2/n2$$

Given *V* and *D*, the effort is calculated as:

$$E = V \times D$$

The unit of measurement of *E* is elementary mental discriminations which represents the difficulty of making the mental comparisons required to implement the algorithm.

McCabe's Cyclomatic Complexity

McCabe's metric [8] is designed to indicate the testability and maintainability of a procedure by measuring the number of "linearly independent" paths through the program. To determine the paths, the procedure is represented as a strongly connected graph with one unique entry and exit point. The nodes are sequential blocks of code, and the edges are decisions causing a branch. The complexity is given by:

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

where

E = the number of edges in the graph

N = the number of nodes in the graph.

According to McCabe, *V(G)* = 10 is a reasonable upper limit for the complexity of a single component of a program.

Structure Metric

It seems reasonable that a more complete measure needs to do more than simple counts of lines or tokens in order to fully capture the complexity of a module. This is due to the fact that within a program, there is a great deal of interaction between modules. Code metrics ignore these dependencies, implicitly assuming that each individual component of a program is a

separate entity. Conversely, structure metrics attempt to quantify the module interactions using the assumption that the inter-dependencies involved contribute to the overall complexity of the program units, and ultimately to that of the entire program. In this study, the structure metric examined is Henry and Kafura's Information Flow metric.

Henry and Kafura's Information Flow Metric

Henry and Kafura [6] [9] developed a metric based on the information flow connections between a procedure and its environment called "fan-in" and "fan-out" which are defined as:

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.

To calculate the fan-in and fan-out for a procedure, a set of relations is generated that reflects the flow of information through input parameters, global data structures and output parameters. From these relations, a flow structure is built that shows all possible program paths through which updates to each global data structure may propagate [10].

The complexity for a procedure is defined as:

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

In addition to procedural complexity, the metric may be utilized for both a module and a level of the hierarchy of the system. Module complexity is defined as the sum of the complexities of the procedures in the module, and the level complexity is the sum of the complexities of the modules within the level.

Hybrid Metric

Since, as stated above, code and structure metrics appear to be measuring different aspects of program complexity, it seems reasonable that a metric be comprised of both types of metrics in order to capture the complexity of the procedure as much as possible. This is what is termed a hybrid metric. More succinctly, a hybrid metric is composed of one or more code metrics and one or more structure metrics. This study examines the hybrid form of Henry and Kafura's Information Flow metric.

Henry and Kafura's Information Flow Metric

The hybrid form of Henry and Kafura's Information Flow metric which was used in an actual study on the UNIX operating system is described in [6]. The formula is:

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

where C_{ip} is the internal complexity of procedure p.

The metric used for the internal complexity C_{ip} may be any code metric.

Description of a Software Metric Analyzer

We have developed a software metric analyzer for use in our research. The analyzer takes as input either the graphical design, textual design, or the source code and produces, as output, a number of complexity metric values. The metric analyzer requires syntactically correct code. When using the analyzer at design time, input consists of syntactically correct graphical designs written in GPL or syntactically correct PDL designs. A general relation language has been successfully used as a tool to express the intermediate form of the design or source code [11]. This intermediate form is then translated into a set of relations which are interpreted to produce metrics. The software quality metric analyzer is based on **LEX** (a lexical analyzer generator) and

YACC (Yet Another Compiler-Compiler), which are tools available with a **UNIX** environment. Hence, the analyzer requires a **UNIX** system.

The remainder of this section describes the details of the implementation of the metric analyzer. For purposes of discussion, the analyzer is divided into distinct three passes. See Figure 3 for a diagram of the analyzer.

A software metric analyzer, which takes as input PDL or source code and produces several software metrics, has been developed for use in our research. A general relation language has been successfully used as a tool to express the intermediate form of the design or source code [11]. This intermediate form is then translated into a set of relations which are interpreted to produce metrics. The software quality metric analyzer is based on **LEX** (a lexical analyzer) and **YACC** (Yet Another Compiler Compiler) which are tools available with a **UNIX** environment. Hence, the analyzer requires a **UNIX** system.

The remained of this section describes the details of the implementation of the software quality metric analyzer. For purposes of discussion, the analyzer is divided into distinct three passes. See Figure 3.

Pass 1

Pass one has as input the Backus-Naur form (BNF) grammar for the source language to be analyzed, the semantic routines which dictate processing for each production in the grammar, and the design or source code to be analyzed. A file containing the intrinsic (built-in) functions, peculiar to the source language is also input. For obvious reasons, these functions should not be treated as real functions; they actually act similar to complicated operators and as such are treated as operators. The source code to be analyzed is assumed to be syntactically correct.

Two files are output from pass one. The first file contains the language dependent metrics for each procedure: lines of code (**LOC**) [12], McCabe's Cyclomatic Complexity (**CC**) [8], and

Halstead's Software Science indicators length, volume, and effort (**N**, **V**, and **E** respectively) [7]. These metrics are produced in pass one since this is the only pass which has the actual code necessary to generate them. The second file output from pass one contains the Relation Language code which is equivalent to the source code. Pass one is the only language-dependent portion of the analyzer. Current source languages processed are the PDL used in this study, Pascal, "C", FORTRAN, and THLL, a language used by the United States Navy. A Pass one for Ada is currently being developed.

A difficult portion of this study was to write a pass one for the graphical language. Since there is no BNF associated with the graphical tool and no prior definition of a line of code, the translation of GPL to relation language code was a challenge. The interested reader is referred to [4] for the exact definition of the translation process.

Pass 2

Pass two use the UNIX tools LEX and YACC. The Relation Language code from pass one is translated into a "set of relations" [10]. This set is completely independent of the original language. Code can be processed one procedure at a time. An advantage is that the Relation Language code for the procedure is the only information necessary to generate its relations. An additional advantage is that source code could be translated into Relation Language code and then analyzed at a separate facility. This feature allows any proprietary details in the original source code to be hidden from the analysis process [11].

Pass 3

Three general classes of software metrics can be distinguished: *structure metrics,* which are measures based on automated analysis of the system's design structure, *code metrics,* which are measures based on implementation details, and *hybrid metrics*, which combine features of both structure and code metrics. As previously proposed by [5], [9], and [1], this research

shows that the structure metrics are global indicators of software quality which can be taken early in the life cycle, while code and hybrid metrics can be brought into use as more implementation details become visible.

Pass three and the associated implementations of the structure metric are written in standard Pascal. The relations file from pass two generates the three structure metrics: Henry and Kafura's Information Flow metric [6], McClure's Invocation metric [13], and Woodfield's Review Complexity metric [14]. Only the Information Flow metric was available for this study. The structure metrics and the code metrics (file one from pass one) produce the hybrid metrics.

A quantitative measurement of design structure can be defined only in terms of those features of the software product which have emerged during the (high-level) design phase. To define a numerical measure, structure metrics use only these features, components, and relationships among components. Note that the actual source code is not necessary to observe the interconnections among components of a system.

A structure measure based on the data relationships among components is the Information Flow metric [15]. This metric identifies the sources (fan-in) and destinations (fan-out) of all data related to a given component. The data transmission may be through global data structure, parameters, or side-effects. The fan-in and fan-out are then used to compute a worst-case estimate of the communication "complexity" of this component. This complexity measure attempts to gauge the strength of the component's communication relationships with other components.

As previously stated, pass three is written completely in standard Pascal and is independent of a UNIX environment. The user is in complete control of the selection of the above metrics to be run and the method of viewing the metrics. The user decides which of the structure metrics he desires to apply to his system. In addition to running the structure metrics and examining them, the user is allowed to define modules (a related collection of procedures) or levels (a related collection of modules). It is assumed that the user would like to view all related procedures as

single module, and likewise, view all related modules as a single level. This feature is especially useful for very large systems. Hardcopies of all reports are available at any time.

III. Complexity Measurement of Designs

Introduction

In this section, 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 the 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 consumable 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 the textual PDL. The assignment required the students to submit an initial design, one week prior to the assignment due date, on the due date, a revised design, and the Pascal source code and simulation results was submitted. 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 the statistical 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 does not have the same number of routines as the source code. Often the source code uses 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.

code. Looking at Table 3, which gives the mean complexities of GPL and the PDL 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 3 and the revised correlations are in Table 2.

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	0.995	0.710	0.952

PDL								
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

Data Presentation Information

Table 1 contains the abbreviations for the code, structure and hybrid metrics that are used in the tables displayed throughout this study. An abbreviation for each of the nine metrics calculated is given.

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 Complexity	INFO-CC

Comparison of Initial and Revised Designs

Comparing the initial and revised designs provides a measure of the change required in order to achieve a working system. This measure is meaningful only if there is a good correlation between revised design and source code. In this study there is a good correlation between revised design and source code, and that correlation is presented in the next section. Table 2 displays the correlations of initial to revised designs for both the GPL and the PDL groups. The correlations between the code metrics in GPL designs are higher than those in PDL 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 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

Table 3. 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

PDL

Design	LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
initial	18.166	118.429	617.606	18489.24	4.20	16456.74	718024	678874000	48765.1
revised	19.536	128.219	675.426	21049.67	4.47	21447.10	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

Regression Analysis of GPL and the PDL

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 [16]. 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 is performed to see if there is any relationship between the two. Table 4 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 5 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. Figure 4 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. 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

Table 4. Correlations of Revised Designs to Designed Source

GPL								
LOC	**N**	**V**	**E**	**CC**	**INFO**	**INFO-L**	**INFO-E**	**INFO-CC**
0.780	0.702	0.660	0.508	0.793	0.808	0.788	0.737	0.752

PDL								
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	0.800

Table 5. Regression Line Equations and Statistics for GPL Design

GPL Regression Line Information				
		Coef	Std. Error	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
Information Flow with Length	Intercept Slope	4985.403 1.278	1709.029 0.082	2.917 15.524
Information Flow with Effort	Intercept Slope	14976060 3.025	4711310 0.229	3.178 13.235
Information Flow with Cyclomatic	Intercept Slope	1284.167 1.539	735.014 0.111	1.747 13.812

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 5.

PDL

A correlation analysis between the PDL revised designs and the PDL designed source code is performed prior to doing the regression analysis to determine if there was any relationship

between the two sets of data. Table 4 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 6 displays the equations that result from the regression analysis of the PDL revised designs and the PDL 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 5 gives a plot of the actual data observations, the regression line and the ninety-five percent confidence lines for the PDL information flow complexity measure. Note, when comparing the GPL and PDL plots that the scales may vary.

Table 6. Regression Line Equations and Statistics for PDL Design

PDL Regression Line Information				
		Coef	Std. Error	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 29.197
Information Flow with Length	Intercept Slope	13294.48 0.48	2605.205 0.019	5.103 25.235
Information Flow with Effort	Intercept Slope	22514560 0.712	5495677 0.036	4.097 19.633
Information Flow with Cyclomatic	Intercept Slope	3076.348 0.501	755.927 0.029	4.07 17.505

The results presented in this section indicate that it is possible, given the complexity of a GPL or a PDL 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 designs used in this study are at a very detailed level of refinement and the accuracy of the equations reflects the detail.

IV. Comparison of Design Tools

In this section a comparison of GPL and the PDL 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 PDL Design Complexities

Table 3 presents the mean complexities of both the revised and initial designs for all of the GPL and PDL data. The complexities of the GPL designs are significantly lower than the PDL 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 PDL. Therefore, lower complexities of GPL designs are not necessarily an indication of less complex systems. Given a GPL and a PDL design for the same system, the GPL design should be measurably lower in complexity.

Comparison of GPL and PDL Designed Source Code

In comparing the GPL and PDL designed source code, the procedures are 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

is defined as the complexity of the entire system. Table 7 presents the average module complexities for both the GPL and PDL designed source code. The complexities of the GPL designed source code are considerably lower than those of the PDL designed source code. Table 8 presents the average ratio of complexity of the GPL designed source to the complexity of the PDL designed source. The ratios of the code 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 show that the structure of the GPL designed source code is nearly six times less complex than the PDL designed source code. Due to the small sample size it is necessary to perform a statistical significance test on the module complexities. Table 9 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 PDL Designed Pascal Source Code

GPL

Module	LOC	N	V	E	CC	INFO
all	332.778	2438.222	12553.444	515028.5	95.000	122431400
request	100.889	749.556	3781.000	147067.3	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
recieve	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
recieve	2.165298e8	2.458338e11	6.926457e7
deadlock	2.472898e6	3.753299e9	6.554804e5

PDL

Module	LOC	N	V	E	CC	INFO
all	412.778	2981.222	15588.110	612871.9	106.000	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
recieve	100.333	740.556	3712.000	49763.6	27.667	6031354
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
recieve	1.097365e9	1.268334e12	2.797951e8
deadlock	1.206509e9	1.238027e12	3.358399e8

Table 8. Average Ratio of Complexities of GPL to PDL Designed Sourc

LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
0.687	0.707	0.696	0.744	0.750	0.194	0.172	0.160	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	0.967	0.413	0.355	1.279	1.321	1.318	1.257

In this section results have been presented that indicate that systems designed in GPL are significantly less complex than systems designed in the PDL 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 PDL designs directly is also shown.

Additional Measurements

In this section the intermetric correlations for the GPL and PDL revised designs and the GPL and PDL designed source code are displayed to replicate the results of several earlier studies [9] [17][2][5].

Intermetric Correlations

Table 10, Table 11, Table 12, and Table 13 present the intermetric correlations for GPL and PDL revised designs and GPL and PDL designed source code. Both sets of GPL correlations and the PDL revised design correlations reflect the results of another study with the code metrics having high correlations to the other code 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 code metrics [9]. These

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

This section shows that the complexity measures of GPL designs replicate the results of another study with the code metrics correlating well with the other code 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 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 Cod

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 PDL Revised Designs

PDL 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	0.695		
INFO-CC	0.486	0.453	0.477	0.317	0.040	0.695	0.900	0.963	

Table 13. Intermetric Correlations for PDL Designed Pascal Source Cod

PDL 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	1000		
INFO-CC	0.682	0.680	0.597	0.385	0.734	0.997	1.000	0.999	

V. Conclusions

This study began to explore the hypothesis that systems designed using a graphical language are less complex than systems designed using a textual design language.

In section three, the results of measuring GPL designs are given. Section three also gave a set of equations to predict, with more than 95% confidence, the complexity of source code. These equations allow the selection of the single least complex design, from a group of designs, that perform the same task and it serves to shorten the design-code-measure-redesign cycle to a design-measure-redesign cycle.

Section four compared GPL designed source code with PDL designed source code. The mean complexities of GPL designed source code are significantly lower than the mean complexities of the PDL 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 the PDL for designing system structure. These results support the hypotheses of researchers who

state that humans think in images and that the human brain works more effectively in images.

In summary, we can:

- Predict the complexity of GPL designed source code with 95% confidence
- Show less complex source code
  - code 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 a PDL for designing system structure.

This research represents a first attempt to quantitatively measure a graphical design and to compare the resultant source code from both a graphical and textual design. Obviously, additional validations of this type are needed to support these results.

Bibliography

[1] Henry, S.M., Kafura, D., "The Evaluation of Software Systems Structure Using Quanitative Software Metrics," Software: Practice and Experience, Vol. 14, No. 6, June 1984, pp. 561-563.

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

[3] Hartson, H.R., Advances in Human-Computer Interaction, Norwood, NJ, Abbex Publishing Company, 1985.

[4] Henry, S.M., Goff, R.A., **Complexity Measurement of a Graphical Programming Language**. Technical Report 87-35, Virginia Tech, Department of Computer Science, 1987.

[5] Canning, J.T., **The Application of Software Metrics to Large-Scale Systems**, Ph.D. Dissertation, Virginia Tech, Computer Science Department, April 1985.

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

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

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

[9] Henry, S.M., Kafura, D.G., Harris, K., "On the Relationships Among Three Software Metrics", **Proceedings of the Sigmetrics Conference on Performance Evaluation Review**, Vol. 10, No. 1, Spring 1981.

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

[11] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers", **Journal of Systems and Software**, 8(3): 3-11; January 1988.

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

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

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

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

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

[17] 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.

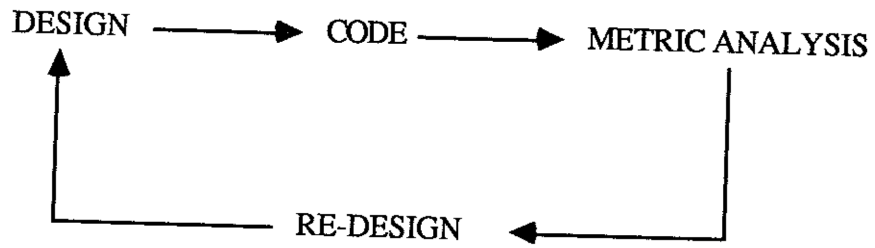

Figure 1. Diagram of Currently Used Software Life Cycle

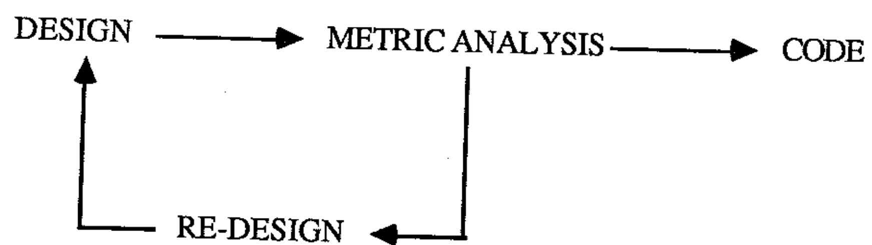

Figure 2. Diagram of Proposed Reduced Software Life Cycle

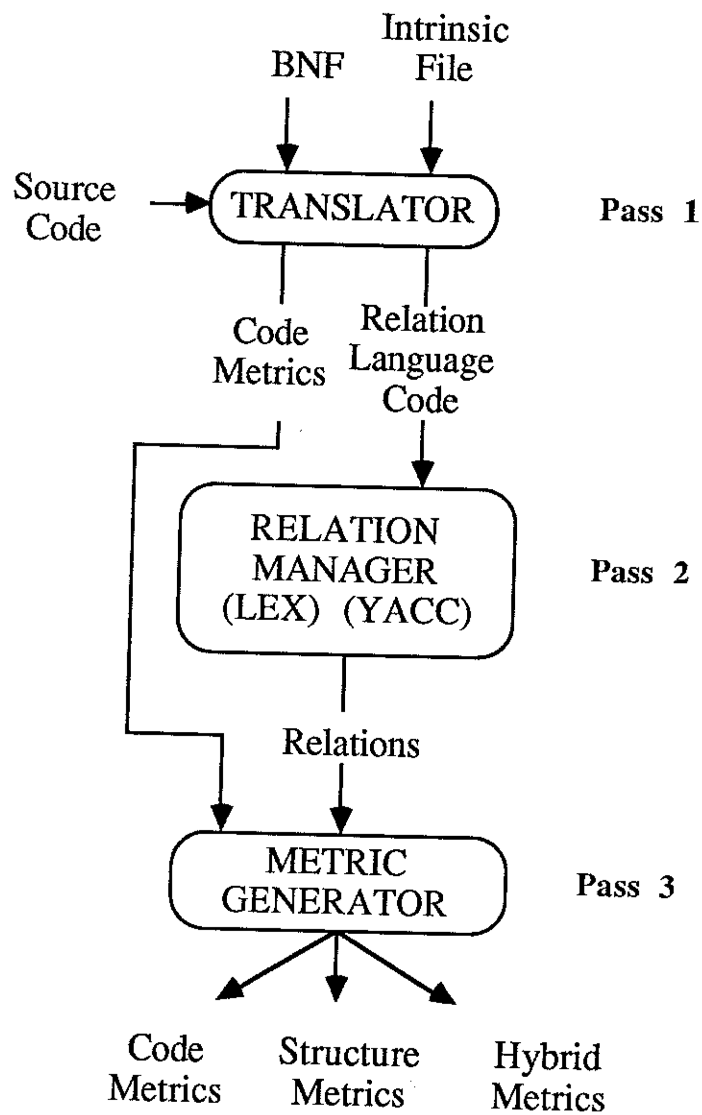

Figure 3. Software Metric Analyzer

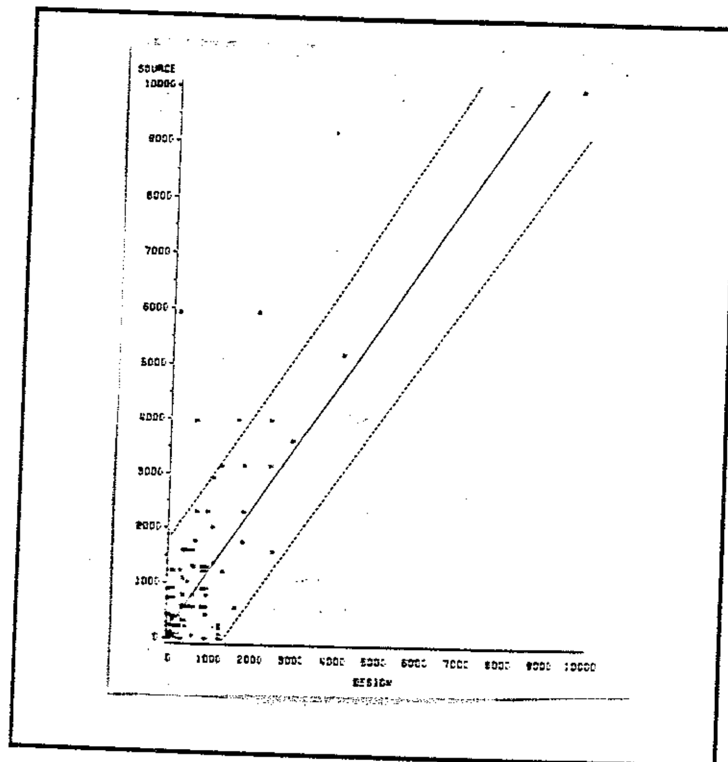

Figure 4. GPL Information Flow Regression and 95% Confidence Lines

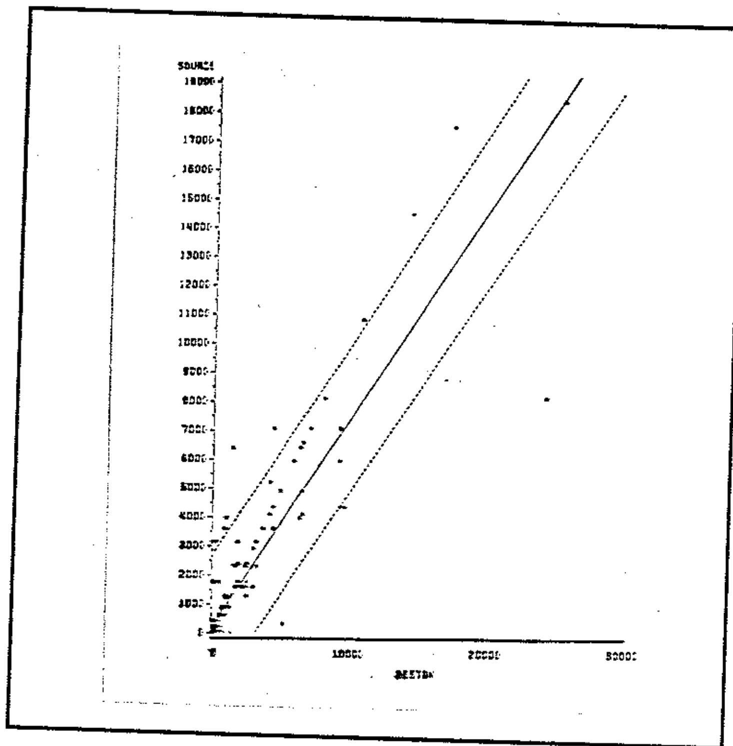

Figure 5. PDL Information Flow Regression and 95% Confidence Lines