Complexity Measurement of a Graphical Programming Language

By Sallie Henry and Roger Goff

TR 87-35

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(11), 1065–1088 (NOVEMBER 1989)

Complexity Measurement of a Graphical Programming Language

SALLIE HENRY AND ROGER GOFF
Computer Science Department, Virginia Tech., Blacksburg, VA 24061, U.S.A.

SUMMARY

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 pre-check on complexity and secondly by measuring the complexity of produced software as a post-check. This research attempts to unify the approach to creating reliable software by providing the ability to measure the complexity of a design prior to its implementation.

We have successfully defined and applied software metrics to graphical designs in an effort to predict software complexity early in the software life cycle. Metric values from the graphical design are input to predictor equations, provided in this paper, to give metric values for the resultant source code.

KEY WORDS Graphical design languages Design methodologies Software complexity metrics Design metrics Henry and Kafura's information flow metric Halstead's software science indicators McCabe's cyclomatic complexity

INTRODUCTION

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 non-programmer managers since there is little syntax to learn and only a small set of symbols with which to become familiar. Graphical design languages (GDLs) are excellent tools for general design, and in contrast to their graphical ancestors are well suited to detailed design. GDLs 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 GDLs in real-world environments.

Many GDLs have been proposed in the literature.[1, 2] The purpose of this study is not to compare GDLs but to apply design metrics to a graphical design to determine the complexity of the resultant source code. Highly complex source code tends to have more errors associated with it and is thus more difficult to maintain.[3–5] A graphical design language, GPL, is used in this research.

0038–0644/89/111065–24$12.00
© 1989 by John Wiley & Sons, Ltd.

Received 23 June 1988
Revised 10 April 1989

GPL is the graphical programming language of the dialogue management system (DMS) being developed at Virginia Tech.[6] 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 SFDs 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 is unique in the sense that it too is a programming language rather than just a design language. The DMS system has a coder under development, which takes the SFDs for a program and generates high-level language code. At some point in the future the coder will generate C code. The DMS project also has plans for a behavioural demonstrator which will dynamically execute a program, while allowing the user to examine and change variable values, and exhaustively test parts of a program.[7, 8]

This research attempts to take designs written in GPL, translate the designs into a form recognizable by our metric tool and finally analyse the designs. By using designs and the source code generated from those designs we can statistically derive prediction equations which take design metric values and produce source code complexity. Metric values for both the graphical designs and corresponding source code were necessary to form the prediction equations. In the future only the metric values for the design and the derived equations are necessary to predict the metric values for the resulting source code. Evaluating a software system at design time can save a large amount of time and money in the production of software.

In general, the software life cycle consists 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 metrics. Figure 1 is a diagram of this part of the software life cycle using software metrics. First, a design is created and implemented in software. At that point, software 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 eliminates 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 with a graphical design tool and to define metrics for a graphical environment.

Determining potential problems early in the life cycle can reduce the cost of software development. For example, if a software component is found to be complex at design time, the designers can re-design the component prior to implementation. At times, some components simply are complex by nature. If that complexity is discovered early, a testing effort may also begin early to fully exercise that component prior to release.

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 complexity of the resultant source code.

The following sections describe the software metrics and associated tools used in this study, the translation of GPL for use by the metric analyser together with a

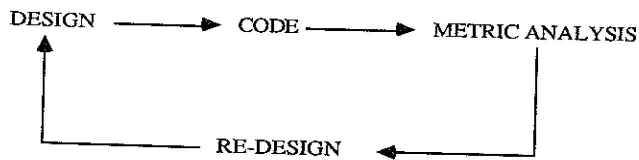

Figure 1. Diagram of currently used software life cycle

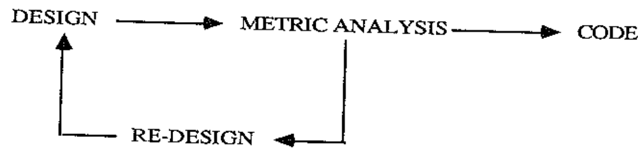

Figure 2. Diagram of proposed reduced software life cycle

definition of metrics for a GDL, the data used in this research together with a statistical analysis of the results and finally our conclusions.

SOFTWARE METRICS AND ANALYSIS TOOL

Software metrics provide a way quantitatively to measure the complexity of software. There are three classifications of metrics that are used to measure the complexity of source code: *code metrics* which measure physical characteristics of the software, such as length or number of tokens, *structure metrics* which measure the connectivity of the software, such as the flow of information through the program and flow of control and *hybrid metrics* which are a combination of code and structure metrics. The metrics are briefly discussed in this section. Interested readers are asked to refer to the references for more details. The remainder of this section describes the tool used to collect the metric values.

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.[3–5, 9, 10]

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[11]) 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

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

The final Halstead metric examined is effort (E). The effort matrix, which is used to indicate the effort of understanding, is dependent on the volume (V) and the difficulty (D). The difficulty is estimated as

$$D = (n_1/2)(N_2/n_2)$$

Given V and D, the effort is calculated as

$$E = VD$$

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[12] 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 is the number of edges in the graph and N is 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 will need to do more than simple counts of lines or tokens in order fully to 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 interdependencies 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[3, 13] 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.[14]

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 used 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 composed 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 hybrid 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 Reference 3. The formula is

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

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 analyser

We have developed a software metric analyser for use in our research. The analyser is provided that takes as input either the graphical design or the source code and produces, as output, a number of complexity metric values. The metric analyser requires syntactically correct code. When using the analyser at design time, input consists of syntactically correct graphical designs written in GPL. A general relation language has been successfully used as a tool to express the intermediate form of the design or source code.[15] This intermediate form is then translated into a set of relations which are interpreted to produce metrics. The software metric analyzer is based on LEX (a lexical analyser generator) and YACC (yet another compiler-compiler), which are tools available with a UNIX environment. Hence, the analyser requires a UNIX system.

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

Pass 1

Pass 1 has as input the Backus–Naur form (BNF) grammar for the source language to be analysed, the semantic routines which dictate processing for each production in the grammar, and the design or source code to be analysed. Current source languages processed are Pascal, C, FORTRAN and THLL, a language used by the United States Navy. A pass 1 for Ada is currently under development. A file containing the intrinsic (i.e. 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 in a similar way to complicated operators and as such are treated as operators. The source code to be analysed is assumed to be syntactically correct. Note that this is the only pass of the analyser that has the source code available to it.

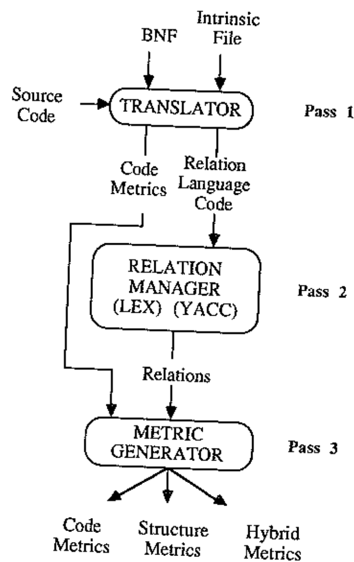

Figure 3. Diagram of the software metric analyser

Two files are output from pass 1. The first file contains the language-dependent metrics (code metrics) for each procedure: lines of code (LOC),[16] McCabe's cyclomatic complexity (CC)[12] and Halstead's software science indicators of length, volume and effort (N, V and E, respectively).[11] These metrics are produced in pass 1 since this is the only pass which has the actual code necessary to generate them. The second file output from pass 1 contains the relation language code which is equivalent to the source code. Pass 1 is the only language-dependent portion of the analyser.

A difficult portion of this study was to write a translator for a graphical language. There is no BNF for graphical symbols, and questions such as 'What is a line of code?' arose. The next section presents our interpretation of pass 1.

Pass 2

Pass 2 uses the UNIX tools LEX and YACC. The relation language code from pass 1 is translated into a 'set of relations'.[14] 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 analysed at a separate facility. This feature allows any proprietary details in the original source code to be hidden from the analysis process.[15]

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,[9, 10, 13, 17] this research shows that the structure metrics are global indicators of software complexity which can be taken early in the life cycle, whereas code and hybrid metrics can be brought into use as more implementation details become visible.

The relation file from pass 2 contains the necessary information to generate the three structure metrics: Henry and Kafura's information flow metric,[3] McClure's invocation metric[18] and Woodfield's review complexity metric.[19] The structure metrics and the code metrics (file 1 from pass 1) are the components of the hybrid metrics. The information flow metric is the only structure metric that was available for this study.

Pass 3 is written completely in standard Pascal and is independent of a UNIX environment. The user is in complete control of the selection of the structure and hybrid metrics to be run and the method of viewing the metrics. 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 a single module, and likewise, view all related modules as a single level. This feature is especially useful for very large systems.

GRAPHICAL LANGUAGE TRANSLATION

It was necessary to develop a translator, pass 1 of Figure 3, to analyse the GPL. This section describes the translation of the graphical language to its analysable form. Translation of GPL is a twofold process. First the graphical design is analysed to generate code metrics used by pass 3 of the analyser. The second process is the generation of relation language code while preserving the control and information flow of the corresponding graphical design. A detailed description of the graphical language, the relation language and the actual translation process is presented.

GPL

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.[6] 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 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. Of interest in this study are the computational development environment and the symbols in GPL used to create the computational portion of a system's design.

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 4 presents the symbols in GPL used for computational design and Figure 5 presents a simple example of a supervisory

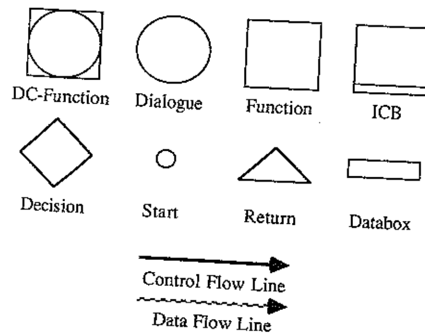

Figure 4. Computational design symbols in GPL

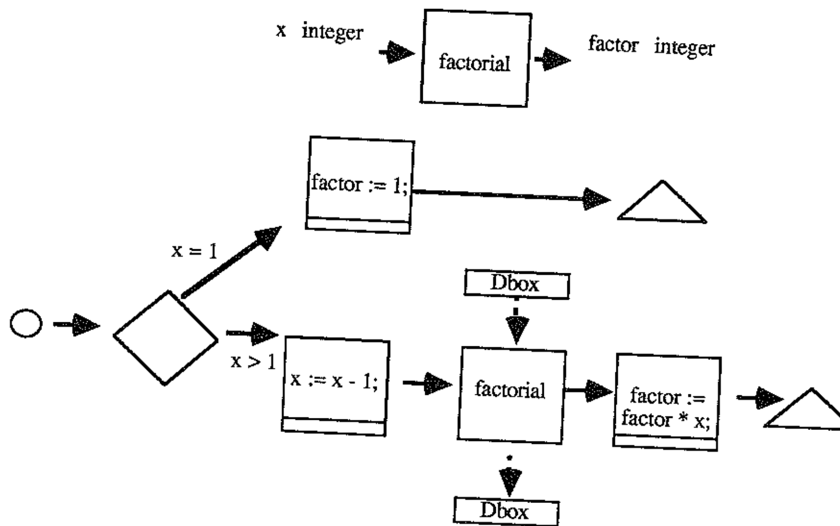

Figure 5. A GPL supervisory cell

cell. The example is the factorial function. The syntax and semantics of each symbol and the information associated with each supervisor is discussed.

The computational design symbols in an 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 labelled $x = 1$ in Figure 5 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 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 labelled *factorial* in Figure 5 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.

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 than 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 receive 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 relation language

The relation language[15] 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 owing to the proprietary nature of the products. Software organizations do not want their algorithms 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 6, 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, owing to the presence of complicated high-level language constructs such as Pascal's with statement and FORTRAN's common block. Variables defined as type struct are treated as global variables, and may be defined anywhere in a program. 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 a 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 statements 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.

Declaration Constructs

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 Constructs

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

Figure 6. Constructs in the relational language

Procedure calls. Procedure calls are identical to those in other high-level block-structured languages such as 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 language 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 it 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

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 straightforward 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 infor-

mation 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. There are three steps to translating a GPL subroutine into relation language: procedure header generation, local variable declarations generation and SFD member generation.

Procedure header generation

Generation of the procedure header involves little processing other than information retrieval. The procedure name and a list of its input and output parameters are retrieved from a database relation. 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 a union on the lists of input and output parameters and generate the resulting set of parameters for the procedure header.

Local variable declaration 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.

SFD member generation

The term 'member' refers to each symbol in an SFD. Many of the member types in an SFD require no code to be generated in the relation language. These member types are 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: function, DC-function, dialogue and ICB.

One might believe that the decision member type ought not to appear with the member types that require no relations language code to be generated due to the existence of the condition statement in the relation language. However, the decision symbol in GPL and the condition statement in relations language 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 affected by the values of 'condition' variables are easily found during visual examination of an SFD. Program constructs such as loops and if . . . then (. . . else)s are quickly recognized. Control flow, however, is difficult to discern given the limited information available in the database which includes the member each arc starts at or 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. There are two steps to translating function, DC-function, dialogue, and ICB member types into relation language: conditional generation and code generation.

Conditional generation

For each of the function, DC-function, dialogue function and ICB member types, 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.

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.

Code generation

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 statement 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 code metric values

Code metric value calculations are performed concurrently with the translation of GPL to relation language and are discussed in the following sections. This research defines the process of applying 'code metrics' to graphical languages.

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

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

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 the 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.

McCabe's cyclomatic complexity

The cyclomatic complexity ($V(G)$) is perhaps the easiest code metric value to calculate. A graph's cyclomatic number is given by

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

where $V(G)$ is the cyclomatic number of a graph, E is the number of edges in a graph and N is 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 an 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 7 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 code metrics to calculate, owing to the ambiguity in determining what is an operand and what is an operator, are Halstead's software science indicators. This study defines the required counts (unique and total operands and unique and total operators) on a graphical language. The counts of operands are straightforward and include anything that appears as text in an 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 then modified by the high-level language translator used to translate the code in the ICBs. The counts of operators in GPL are less intuitive than those 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, which are performed by the high-level-language translator, are accumulated with

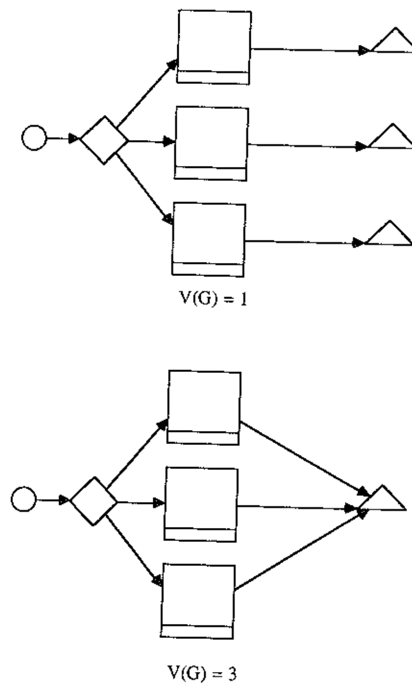

Figure 7. Multiple exit point handling for cyclomatic complexity calculation

those returned by the conditional parser, along with identification of operators being performed as in conventional, textual languages. The number of members in an SFD and the number of control and data flow arcs in an 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.

METRIC ANALYSIS

In this section, the results of analysing 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 source code corresponding to a GPL design to be predicted.

In this experiment, all of the designs are at a fairly detailed level for two reasons. First, we wanted to determine if our definitions of applying metrics to a graphical design were correct. Could the values of the metrics at design time predict the complexity of the source code? Secondly, a previous study determined that code metrics had little or no value when applied at the architectural design level. However, the structure metric seemed to give a good indication of source-code complexity early in the design phase.[10]

The experiment

The data in this study were collected from an assignment given in a graduate-level operating systems course. 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.[20] 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. The revised designs were included as a part of the assignment.

Data preparation

In order to perform statistical analysis of the correlations of initial design to revised design and of revised design to source, and to perform the regression analysis, it is necessary to have the same number of data observations (i.e. 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 may be 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 analysed. When a routine exists in the source and does not have a corresponding routine in the design, one sums 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 procedures, 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 (e.g. GPL or 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 were used in this analysis.

GPL measurement and predictor equations

Table I 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 in order to achieve a working system. This measure is meaningful only if there is a good correlation of the metric values between revised design and source code. In this study there is a good correlation between revised design and source code. Table II

Table I. 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

Table II. Correlations between initial and revised designs

LOC	*N*	*V*	*E*	CC	INFO	INFO-L	INFO-E	INFO-CC
GPL								
0·884	0·917	0·972	0·942	0·935	0·877	0·995	0·710	0·952
GPL design correlations with no outliers								
0·905	0·943	0·974	0·978	0·944	0·924	0·829	0·926	0·865

displays the correlations of initial to revised designs for GPL. The correlations between the code metrics in GPL designs are very high. Looking at Table III, which gives the mean complexities of GPL, it is interesting to note that the complexity of GPL designs actually went down from the initial to the revised designs for the structure and hybrid metrics. Further investigation revealed that the complexities of two procedures 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 III and the revised correlations are in Table II.

Regression analysis of GPL

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.

A correlation between the two sets of values was performed to see if there was any relationship between the two. Table IV gives the results of the correlation. These correlations are 'high', but certainly not considered 'very high'. This implies that many changes or revisions occured prior to actual implementation. However, our priority is not to discuss the fact that designs correlate with source but to provide a model which

Table III. Mean complexities of initial and revised designs

Design	LOC	N	V	E	CC	INFO	INFO-L	INFO-E	INFO-CC
GPL (all procedures)									
Initial	14·032	85·95	251·656	6374·66	3·552	14,129·62	1,362,948	808,062,000	337,889
Revised	14·519	87·89	261·617	6442·96	3·643	1003·13	28,337	18,113,410	8164
GPL (with outliers removed)									
Initial	13·375	80·678	233·974	5698·52	3·38	688·395	10,431·23	7,348,219	2930·033
Revised	14·151	84·618	246·842	6009·91	3·52	686·559	12,050·82	8,206,303	3302·488

predicts the source code complexity from the design. If at design time, one discovers a high complexity (in one or more of the metrics) and a high correlation between designs and source code complexity has been proved, then one can detect probably problem areas at design time.[10]

A less detailed design still has a reasonable correlation with the structure metric but has very low correlations with the code metrics. High complexity with a high-level design still indicates potential problems but low complexities in a high level design do not necessarily imply low complexities in the corresponding source code.[10]

The high correlations in Table IV indicate that a relationship between the data sets may exist, so the regression analysis is performed. Table V contains the equations that result from the regression analysis of the GPL revised designs and the GPL designed source code. The 'Coefficient' column gives the value of the y-axis intercept and the slope of the regression line for the corresponding metric. The 'Standard error' column gives the standard error found in the calculation of the coefficient and the 't-value' column 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 twice the standard error. A t-value of greater than two generally represents 95 per cent confidence that the corresponding coefficient is correct. The t-values for each of the metrics' slopes are well above 2, and 99 per cent confidence in their values can easily be assumed. Ninety-nine per cent 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 8 gives a plot of the actual data observations, the regression line and the ninety-five per cent confidence lines for the GPL information flow measure. The prediction equation for a procedure's information flow complexity is as follows:

$$y = 1{\cdot}103x + 205{\cdot}167$$

Table IV. Correlations between revised designs and 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

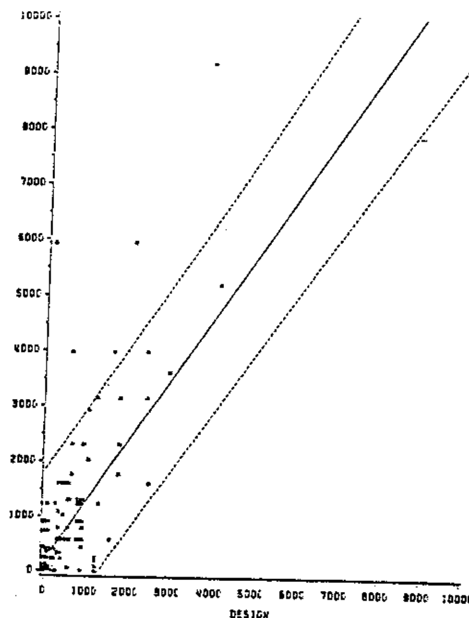

Figure 8. Regression line and 95 per cent confidence lines for information flow complexity in GPL

where y is the predicted source code information flow complexity of the procedure and x is the calculated design information flow complexity of the procedure. With 95 per cent 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 V. Figures 9 and 10 graph the GPL length and effort regressions, respectively.

The results presented in this section indicate that it is possible, given the complexity of a GPL, 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. A similar research project using a textual design language found that with a less detailed level of refinement, the confidence level in the prediction equations decreased.[10] Although this result seems obvious, this research concentrated on defining metrics for a graphical language and proving that the metrics could indeed predict source code complexity. Our belief is that prediction equations could be calibrated for less detailed designs, but the confidence level for those equations would decrease.

CONCLUSIONS

This study was begun with a twofold purpose. The first purpose was to define whether the complexity of a graphical language could be measured, using established software

Table V. Regression line equations and statistics for GPL design

		Coefficient	Standard error	t-value
Length	Intercept	3·878	0·871	4·454
	Slope	0·826	0·055	15·105
N	Intercept	22·258	8·781	2·535
	Slope	1·141	0·096	11·947
Volume	Intercept	211·697	42·367	4·997
	Slope	1·639	0·154	10·643
Effort	Intercept	12892·83	2666·116	4·836
	Slope	2·222	0·311	7·15
Cyclomatic	Intercept	−0·03	0·334	−0·09
	Slope	1·325	0·084	15·799
Information	Intercept	205·167	77·814	2·637
Flow	Slope	1·103	0·066	16·629
Information flow	Intercept	4985·403	1709·029	2·917
with length	Slope	1·278	0·082	15·524
Information flow	Intercept	14,976,060	4,711,310	3·178
with effort	Slope	3·025	0·229	13·235
Information flow	Intercept	1284·167	735·014	1·747
with cyclomatic	Slope	1·539−	0·111	13·812

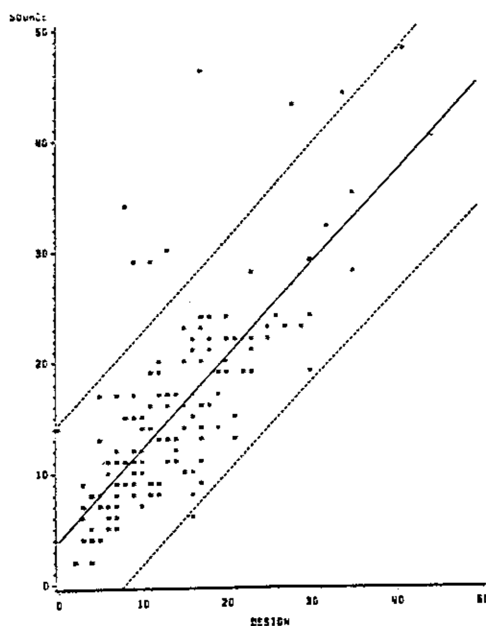

Figure 9. GPL length regression and 95 per cent confidence lines

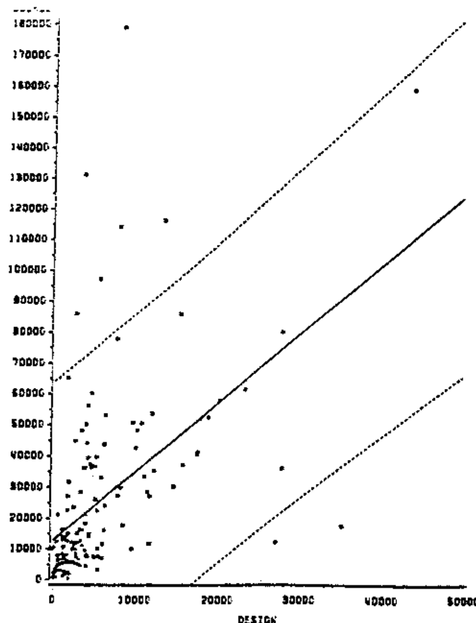

Figure 10. GPL effort regression and 95 per cent confidence lines

metrics. The second was to show that software metrics may be applied early in the life cycle to predict resultant source code complexity.

In a prior section 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 metric analyser draws the information needed to produce the measures. The results of measuring GPL designs and a set of equations to predict, with more than 95 per cent confidence, the complexity of source code was discussed. These equations allow the selection of the least complex design from a group of designs that perform the same task, and serve to shorten the design–code–measure–redesign cycle to a design–measure–redesign cycle.

This research is an initial attempt which indicates the ability to measure graphical designs and to predict source code complexity from GPL designs. As with all metric validation studies, these results need to be duplicated and extended. First, the extension of the metric definition to an architectural, or higher, level of design must be considered. Our inclination is that the results of such an experiment would indicate that the structure metrics are more indicative of source code complexity at this high level. Prior research has shown that code metrics become more reliable as more detail is provided in the design.

A second extension of this research is to validate the results on a 'real' system. This validation presents more difficulty for academics to accomplish. However, since GPL is similar to most production graphical tools, we would expect these results to be duplicated.

Since this initial study was successful, one would hope that additional validation would also be a success. In the future, we hope to extend this research to calibrate prediction equations for less detailed levels of design and to attempt this analysis in a production system.

REFERENCES

1. M. Moriconi and D. Hare, 'Visualizing program designs through Pegasys', *IEEE Computer,* August 1985.
2. E. Glinert and S. Tanimoto, 'Pict: an interactive graphical programming environment', *IEEE Computer*, November 1984.
3. S. M. Henry and G. D. Kafura, 'Software structure metrics based on information flow', *IEEE Trans. Software Engineering*, September (1981).
4. James T. Canning, 'The application of micro and macro software metrics to large scale systems', *Masters Thesis*, Virginia Tech, Department of Computer Science, November 1982.
5. D. Kafura and G. R. Reddy, 'The use of software complexity metrics in software maintenance', *IEEE Trans. Software Engineering,* **SE-13**, (3), (1987).
6. H. R. Hartson, *Advances in Human–Computer Interaction*, Ablex Publishing Company, Norwood, NJ, 1985.
7. J. E. Callan, 'Behavioral demonstrations: an approach to rapid prototyping and requirements execution', *Masters Thesis*, Virginia Tech, Department of Computer Science, 1985.
8. E. C. Smith, 'System support for design and development environments', *Masters Thesis*, Virginia Tech, Department of Computer Science, 1986.
9. J. T. Canning, 'The application of software metrics to large-scale systems', *Ph.D. Dissertation*, Virginia Tech, Computer Science Department, April 1985.
10. C. L. Selig, 'ADLIF — a structured design language for metric analysis', *Masters Thesis*, Virginia Tech, Department of Computer Science, August 1987.
11. M. Halstead, *Elements of Software Science*, Elsevier North Holland, Inc., New York, NY, 1977.
12. T. McCabe, 'A complexity measure', *IEEE Trans. Software Engineering,* December (1976).
13. S. M. Henry, G. D. Kafura and K. Harris, 'On the relationships among three software metrics', *Performance Evaluation Review,* **10**, (1), (1981).
14. D. Kafura and S. Henry, 'Software quality metrics based on interconnectivity', *J. Systems and Software,* **2**, (1982).
15. S. M. Henry, 'A technique for hiding proprietary details while providing sufficient information for researchers', *Journal of Systems and Software,* **8**, (3), 3–11 (1988).
16. S. D. Conte, H. E. Dunsmore and V. Y. Shen, *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1986.
17. S. M. Henry and D. Kafura, 'The evaluation of software systems' structure using quantitative software metrics', *Software — Practice and Experience,* **14**, (6), 561–573 (1984).
18. C. McClure, 'A model for program complexity analysis', *Proceedings Third International Conference on Software Engineering*, Atlanta, GA, May 1978, pp. 149–157.
19. S. Woodfield, 'Enhanced effort estimation by extending basic programming models to include modularity factors', *Ph.D. Dissertation*, Purdue University, Computer Science Department, 1980.
20. M. Maekawa, A. Oldenhoeft and R. Oldenhoeft, *Operating Systems: Advanced Concepts*, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1987.