# A Technique for Hiding Propietary Details While Providing Sufficient Information for Researchers or Do You Recognize This Well-Known Algorithm?

*Sallie Henry*

TR 86-38

# A TECHNIQUE FOR HIDING PROPRIETARY DETAILS WHILE PROVIDING SUFFICIENT INFORMATION FOR RESEARCHERS

or

# DO YOU RECOGNIZE THIS WELL-KNOWN ALGORITHM ?

by

DR. SALLIE HENRY

Department of Computer Science
VIRGINIA POLYTECHNIC INSTITUTE

Blacksburg, VA 24061

## ABSTRACT

A major problem facing software engineering researchers is the difficulty of performing validation experiments requiring "real-world" data. This data may be composed of the requirements document, design descriptions, source code, test plans, and/or error histories. Most major software organizations object to giving this usually proprietary data to researchers. This objection is understandable since these organizations do not desire their proprietary source code or error history to be made public. This paper describes a technique to translate source code into a meta-language which will maintain the proprietary nature of the original source code by hiding the algorithms and data structures employed in the source code. This technique, however, preserves sufficient data for validation experiments to be performed by some software engineering researchers, particularly those in the software quality metrics community.

## I. INTRODUCTION

Can you recognize the algorithm in Figure 1?

This paper explains a technique which could be used to translate software systems into a *sanitized* string of characters. This string is meaningless to anyone attempting to discover the implementation (algorithms, data structures, etc.) of the original system. However, this string can then be processed to obtain information required by some software engineering researchers to evaluate proposed tools and methodologies. In particular, those techniques dealing with software quality metrics.

This encoding technique is illustrated in Figure 1, where a well-known algorithm has been encoded in the meta-language described in this paper. Encryption, in a very simple form, is used to code the identifier names in Figure 1. This simple encryption is to number the identifiers (var1, var2,...) and precede each identifier with the function name (F). This paper is not attempting to evaluate encryption techniques, but simply to suggest that encryption should be used on identifier names when attempting to encode source code.

The perceived use of this encoding technique is to provide validation capabilities on proprietary corporate data. Software engineers propose tools and techniques to aid software developers in producing more reliable software systems. In order to judge the *goodness* of a tool, the system resulting from using the tool must be validated for attributes of *goodness*. For example, a software engineer proposes a new tool to aid in detailed system design. The design document together with the final product and perhaps a problem report history would be required to sufficiently perform a validation on the proposed tool. If the problem reports indicated, for example, that the design tool relieved some of the burden of communication difficulties between the designer and the programmer, then a point of *goodness* may be attributed to the tool.

```
FUNCTION F (FVAR1);

   LOCAL FVAR2, FVAR3;
  BEGIN
     FVAR2 := FVAR1;
     COND  FVAR2 & 100
        BEGIN  WRITELN ('100');
              COND  FVAR2 & 100
                BEGIN  F := FVAR2;
                       FVAR3 := 100;
                      COND  FVAR2 & 100
                        BEGIN  FVAR3 := FVAR3 + FVAR2;
                               FVAR2 := FVAR2 + 100;
                        END;
                      F := FVAR3;
                END;
     END;
  END;
```

Figure 1.   Example of a well-known algorithm

At times, tools and methodologies are proposed and even used without sufficient validation. This lack of validation could result in a designer overlooking a perhaps better tool since that tool's *goodness* has not been *proven*, i.e. the designer could use a bad tool. These methodologies or tools are difficult to evaluate without the supporting documentation.

A major problem facing researchers in the software engineering field is the availability of *real* software systems to validate proposed tools and methodologies. This problem is most prevalent for a researchers in academia. Non-academic software researchers normally have access to some in-house software systems for purposes of validations. However, we should note the weakness of this approach; that in-house systems used for validation purposes lack the commonality of validation results across the industry. Analogously, academicians occasionally must use student projects for preliminary tests on a proposed tool or technique. Some researchers have expressed concern that tools validated in a classroom environment may not produce similar results in an actual production software development environment.

The author's area of research is software quality metrics. Metrics attempt to quantitatively measure the quality of a software system. The most simple metric is lines of code; i.e. a shorter routine has better quality than a longer routine. Regardless of the metric employed, a number representing the "quality" of a piece of software is assigned to a software component. How do we know that these numbers adequately represent the quality of that component? One method is to focus on factors such as structure, readability, etc. and subjectively judge, using these factors, the quality of a routine. A less subjective method of validation is to judge a component's quality by considering the number and severity of errors encountered in that component. Certainly, one attribute of quality is the lack of errors.

Using student projects to perform a validation similar to the above generates very weak results, the experiment is too controlled. True production systems and their associated error histories are essential to validating this type of measurement.

Most organizations classify their software as proprietary and are reluctant to release these production systems. This paper proposes a technique for hiding implementation details while providing sufficient information for, at least, metrics researchers. Other researchers should also benefit from these encoded systems and perhaps will consider using software quality metrics for validating their methodologies.

Section 2 describes our research in software quality metrics This description is included to provide the reader with an explanation of the metrics collected and to supply a flavor of the information required by the metrics. A reader may choose to bypass this section on the first reading of this paper. Section 3 demonstrates the translation of source code to our encoding language. A description of this translation technique as applied to our research is presented in section 4. Finally, section 5 contains our conclusions.

## II. Overview of Software Quality Metrics

One of the basic objectives of software engineering is to transform the creation of software systems from an artistic, poorly understood, and even undisciplined, activity into a carefully controlled, methodical, and predictable enterprise. To make software an engineerable product, it is important that the designers, implementors and maintainers of software systems be able to express the characteristics of the system in objective and quantitative terms. Quantitative measurement is commonly used to evaluate some software characteristics. For example, software performance [LYNW 81] is measured quantitatively in terms of running time, response time, space, etc., while software reliability [MUSJ 75] is measured by mean-time-between-failure, mean time to repair an error, or error density. Other characteristics which have not been satisfactorily quantified are those relating to software quality. Quantitative measures of quality are collectively referred to in this section as software metrics.

When applied to a software system, the word *quality* brings to mind a large range of attributes. Many of these attributes refer to the *process* by which the software system was constructed. Measures of such attributes are, therefore, termed process metrics. For example, the average staffing size, or the average number of years of experience of the staff are process metrics. Alternatively, other quality attributes refer to the software product itself and measures of these attributes are termed product metrics. The software product, and its quality attributes, may be viewed from two different perspectives. One perspective is from the point of view of the users of the software system. Attributes of interest to this group include such factors as ease of use, relevance to the user's applications, cost, and readability of documentation. Another perspective is taken by the personnel who are concerned with design, implementation, and maintenance of the software product and are not themselves the end users. Of interest to this group are such attributes as ease of maintenance, simplicity of code, ease of understanding, and modularity. The metrics discussed in this section fall only in the last of these categories: product metrics which refer to attributes of interest to software designers, implementors, and maintainers. For example, the number of decisions per line of code, or the average number of lines of code per module are product metrics which are included in this class. In the balance of this section the terms *software metrics* and *software product metrics* will be used interchangeably to refer to measures of those attributes relevant to the personnel involved in creating or maintaining the software system.

Three general classes of software metrics can be distinguished: measures based on an automated analysis of the system's design structure, termed *structure metrics;* measures based on implementation details, termed *code metrics;* and measures termed *hybrid metrics,* which combine features of both of these other two. 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 the implementation details become visible.

Our attention should also focus on the process of generating software quality metrics for a software system. In order for software systems to be analyzed, a tool must be developed to automate this analysis. The manual generation of software quality metrics is not feasible for most software systems.

The author has had considerable experience in attempting to completely automate the metric analysis. The major difficulty encountered with this process is the dependency of the analyzer on the source language. In previous versions of the metric analyzer, the process of analyzing a new source language required the development of a preprocessor for the language and the effort expended was approximately one year. The author has recently developed a *language independent* software quality metric analyzer which is discussed in section four of this paper.

## STRUCTURE 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. Many design methodologies, typified by Ross' SADT [ROSD 77] and Yourdan and Constantine's SA-SD [YOUE 79], focus on identifying the components in the system and specifying how these components are related through control and/or data connections.

Accordingly, structure metrics use only these features, components and relationships among components, to define a numerical measure. 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 [HENS 79]. 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 structures, 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.

Another structure metric, the *invocation complexity,* was defined by McClure [MCCC 78]. The relevant features in analyzing the invocation complexity of a components are: (1) all variables which control (via conditional or iteration statements) the invocation of the component; and (2) all components which can affect the value of these variables. In this metric a small invocation complexity is assigned to a component which, for example, is invoked unconditionally by only one other component. A higher complexity is assigned to a component which is invoked conditionally and where the variables in the condition are modified by remote ancestors or descendents of the component.

A third structure metric, defined by Woodfield, is based on the concept of *review complexity* [WOOS 80]. Woodfield observes that a given component must be understood in each context where it is called by another component or affects a value used in another component. In each new context the given component must be reviewed. Due to the learning from previous reviews, each review takes less effort than the previous ones. Accordingly, a decreasing function is used to weight the complexity of each review. The total of all of these weights is the measure assigned to the component. Woodfield applied this measure successfully in a study of multi-procedure student programs.

## CODE METRICS

As is implied by the name, code metrics refer to measures that are defined in terms of features not visible until much of the detailed coding of the system has been completed. The simplest of all of the code metrics is the ubiquitous "lines-of-code" measure. This measure has been used as a common measure of size (e.g., a 100,000 line FORTRAN system), has been used as the basis of programmer productivity measures (e.g., 10 debugged statements per day), has been adopted as a quality criteria (no module should exceed 50 lines of code), and has been a common parameter in software cost estimation models (e.g., in Putnam's model [PUTL 78] or in Boehms's model [BOEB 81]) where completion time and staffing levels are a function of the size of the project measured in lines-of-code. Despite wide differences in the method of counting lines of code, the measures have been found to be no less valid a size measure than other more sophisticated size measures.

Two other well known code metrics are those defined in Halstead's software science [HALM 77] and the measure of cyclomatic complexity defined by McCabe [MCCT 76]. The software science measures are all based on four terms: the number of unique operators and operands, and the number of total operators and operands. The cyclomatic complexity, on the other hand, is simply a count of the number of decision points in structured programs. McCabe has shown that this count is related to the cyclomatic number of the graph which represents the control flow structure of the code being measured. McCabe related this value to the ease with which code could be tested.

Several carefully designed experiments have shown that meaningful relationships exist between these metrics and significant software characteristics [CURB 79] [BASV 83]. Properly used, these metrics can play a useful role during the later phases of the software life cycle (testing, acceptance, maintenance, etc.). More recently a study has shown that a variation of the software science techniques can be used to identify error-prone components [SHEV 85].

The difficulty with all code metrics is that they are available late in the life-cycle. Once the code for a large system has been created there is little incentive -- and usually no available resources -- to undertake a major redesign or reimplementation of the system. In addition, the software science approach has been the subject of considerable controversy. Serious objections have been raised against the experimental design and statistical treatment of a number of the software science experiments [HAMP82] [LASJ 81] [SHEV 83].

## HYBRID METRICS

A third category of software metrics contain what are termed hybrid metrics. Each of these hybrid measures is a modification of one of the structure metrics. A hybrid metric determines a measure for a component by weighting the structure measure for that component by a code measure. To simplify matters we have used only lines-of-code as the weighting term. Using any of the other code metrics does not appear to change the overall character of the results. It should be noted that two of the metrics used in this project (Henry and Kafura's information flow, and Woodfield's review complexity) were originally posed by their authors as hybrid metrics. The two hybrid metrics which have been studied are: information flow weighted by lines-of-code, and Woodfield's metric weighted by lines-of-code. The McClure invocation complexity metric does not lend itself to such weighting.

## VALIDATIONS

The validation of a software metric is vital because it establishes the relationships between the measures of software products (e.g., a complexity metric) and important cost drivers in the software process (e.g., errors, coding time). One form of a validation, which may be termed a quantitative validation, typically involves the empirical analysis of historical project data. Equally important are qualitative validations which seek to demonstrate that the metrics of software are consistent with the prevailing of high quality software design and implementation. Qualitative validations rely on subjective evaluation to address those issues of structure and design which cannot adequately be studied by an examination of historical data. As described below, both quantitative and qualitative validations have been employed in this research area.

Several validations relating to code metrics have been referenced above. These validations illustrated significant relationships exist between the cyclomatic complexity and program comprehensibility, and also between the software science measures and the error-proneness of components.

A few examples of such validations are presented, not as a complete set of metric validations, but as examples of "real world" systems used in metric validations. Indicative of the utility of the structure metrics was a study of the kernel of the UNIX operating system which showed:

> there is a high statistical correlation between the information flow metric and changes to the UNIX kernel [HENS 79] [HENS 81a] [KAFD 82].
>
> information flow metric was statistically different than the code metrics - indicating that it was capturing a distinct feature of the software than the code metrics [HENS 81b].
>
> the metrics could be used to identify, diagnose, and repair a number of structural defects in the software including missing levels of abstraction, improper functional decomposition, and inadequate refinement [HENS 84].

These results have been confirmed by a recently completed comprehensive, retrospective validation employing all four structure metrics. An automated analysis system was written to compute these metrics from the source code of completed systems and, in one case, the skeletal code of a system under design. The two major parts of this comprehensive validation served to confirm the earlier experiments using the information flow metric and to advance our understanding of how several metrics can be together to form a more complete view of the system structure.

Another comprehensive validation study [CANJ 85] examined three systems which were developed at NASA/Goddard by the Computer Science Corporation under the auspices of the Software Engineering Laboratory. Historical data was collected during the development of these systems [NASA 81]. The source code for these systems and the historical data, recording manpower and error data, was used in conjunction with an automated analyzer to determine the statistical relationships between a battery of software metrics and the error/effort characteristics of the systems. This validation study is more robust than those previously attempted in this field. This study:

confirmed the statistical difference between structure metrics and code metric.

showed that structure metrics are competitive with code metrics in their relationship to error occurrences and coding times.

indicated that none of the metrics are precis enough to finely distinguish between component with approximately the same metric readings but that when grouping techniques are used a clearer relationship to errors and coding times emerge.

demonstrated that the grouping techniques cobe used to isolate those components with the worst error and/or coding times characteristics.

The results from several validation experiments were summarized. Also a brief sampling of data from one of these experiments was presented. In general these experiments have shown that significant relationships exist individually between the software metrics and quality attributes such as error characteristics, comprehensibility of code, length of coding time, and structural soundness.

Software quality metrics are used by a wide variety of software engineers to validate design methodologies and various other tools. Section three  describes a language which can be used to encode source code while preserving the fundamental structure of the system necessary to generate the structure metrics.  A tool to generate metrics from the encoded software is explained in section four.

## III. DEFINITION OF RELATION LANGUAGE

Figure 2 displays the relationship between the company with its proprietary software systems and the outside world with its respective encoded software system. This encoding process involves the two steps displayed in Figure 2. The first step encodes the actual source code using the relation language described in this section. An equally important step in this translation process is providing the outside world with some type of historical data base of error information corresponding with the given software system. The names in the data base would be changed to correspond with the names used in the relation language.

The relation language was developed as an intermediate form between the source code and a set of relations [HENS 84]. Relations show the flow of control and flow of information, both of which are necessary to model the structure of the system. These relations together with a set of code metrics are combined to produce the structure metrics defined in the previous section. The process of analyzing source code and producing the corresponding software quality metrics is discussed in the next section.

Data required to generate the structure metrics is dependent on the source code being processed. The relation language is our attempt to first quantify a measurement of the source code independent of the source language and second, to provide software organizations with a vehicle to facilitate data sharing together with the hiding of proprietary data.

To achieve independence among source languages, certain observations were made. All block structured languages have several constructs in common. For example, all source languages in consideration contain assignment statements, iteration statements, conditional statement, sequence, procedure calls, etc. Languages which have been translated are Pascal and THLL (a high level language used by the Naval Surface Weapons Center), with 'C' and FORTRAN soon to follow. Note that we are not interested in languages not suited to this model (LISP, PROLOG, etc.). Instead, we are interested in production software systems typically implemented in languages with the above constructs.

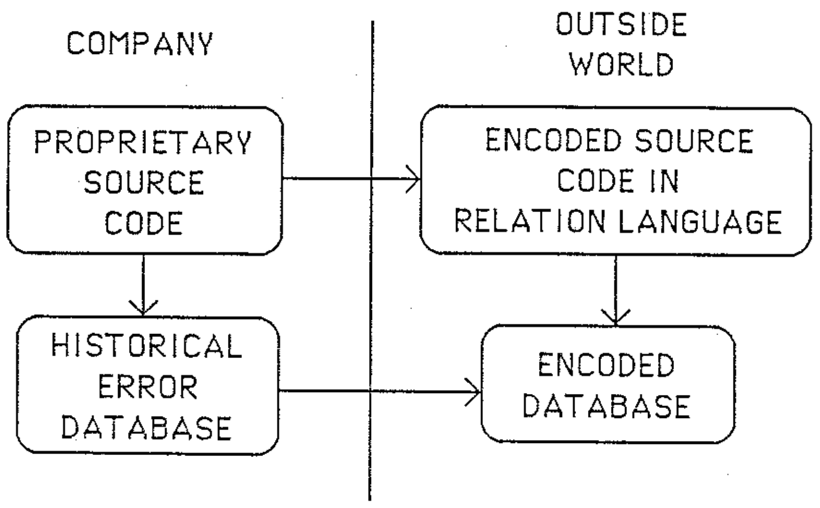
COMPANY
OUTSIDE WORLD
PROPRIETARY SOURCE CODE
ENCODED SOURCE CODE IN RELATION LANGUAGE
HISTORICAL ERROR DATABASE
ENCODED DATABASE

Figure 2. Relation between a company and the outside world

## RELATION LANGUAGE STATEMENT TYPES

The statement types in the relation language are divided into two types: declaration and executable statements. An explanation of the relation language is given with examples of Pascal code being translated. Pascal is simply an example source code. All high-level block-structured languages could be translated in a similar fashion.

### DECLARATION STATEMENTS

Seven keywords are used to describe the declarations.

LOCAL <variable name>;

Any variable declared to be local to a procedure is declared with a LOCAL statement. No type is associated with the variable in the declaration.

For example,

Pascal:

```
TYPE node = RECORD
              x : INTEGER;
              y : ARRAY [1..10] OF CHAR;
            END;

 VAR t   : node;
     s,r : CHAR;
```

Relation Language:

```
LOCAL t;
LOCAL s;
LOCAL r;
```

STRUCT < variable name>;

The STRUCT statement is used to define any non-local variable. In other words any variable referenced in a procedure which is not declared in that procedure is declared with the STRUCT statement.

CONST < variable name>;

This statement is used for any user-defined constants. Note that the value of the constant is not used in the statement.

For example,

| Pascal: | Relation Language: |
|---|---|
| CONST x = 26; | CONST x; |

PROCEDURE < variable name> ( parameters);

A procedure declaration appears just as it would in Pascal with the elimination of the TYPES associated with the parameters. The parameters are separated by commas.

For example,

Pascal:

```
PROCEDURE x ( y : INTEGER; VAR z : node);
```

Relation Language:

```
PROCEDURE x (y,z);
```

FUNCTION < variable name> ( parameters);

A function declaration appears just as it would in Pascal with the elimination of the TYPES associated with the parameters and the return value. Commas separate the parameters.

For example,

Pascal:

```
FUNCTION x ( y : INTEGER; VAR z : node): INTEGER;
```

Relation Language:

```
FUNCTION x (y,z);
```

EXTERNAL < variable name> ;

The EXTERNAL statement was developed for those languages that allow external procedure declarations.

INTRINSIC < variable name> ;

This statement defines a built-in function present in the source language. Essentially, the INTRINSIC statement provides a means to distinguish built-in functions from user-defined functions.

For example,

Pascal:

the SIN function

Relation Language:

```
INTRINSIC SIN
```

## EXECUTABLE STATEMENTS

Executable statements in the source code are translated into the following relation language statements. Three immediate translations are defined.

1. All constants, either character strings or numeric constants are translated to "100".
2. All variables in relational expressions are separated with an "&".
3. All arithmetic operators are translated to "+".

For example,

| Pascal: | Relation Language: |
|---|---|
| x := x * 6 / y; | x := x + 100 + y; |

COND <expression> <stmt> ;

All conditional statement are translated to COND, and the <expression> variables are listed according to rule 2. Pascal conditionals include IF, FOR, WHILE DO, REPEAT UNTIL, and CASE. No differentiation is made between the statements in a THEN and the statements in an ELSE.

For example,

Pascal:

```
IF (( a = b ) and ( c > d + 6 ))
 THEN x := 12
 ELSE y := 0;
```

Relation Language:

```
COND a & b & c & d  & 100
BEGIN
    x := 100;
    y := 100;
END;
```

Note, as in Pascal, **BEGIN END**'s are used to group statements.

Figure 3 summarizes the relation language statements.

**Declaration statements**

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

**Executable statements**

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

Figure 3. Relation Language Statements

## ACTUAL TRANSLATION PROCESS

All high-level, block-structured source languages can be translated into this relation language. The major difficulty encountered in this translation process is specifying variable names. Resolving variable identification is a difficult problem in the translation process. Each language has some idiosyncrasies regarding variable identification. The Pascal **WITH** statement is an example of one of these idiosyncrasies. Not all variables used in a **WITH** block need to be modified by the qualifier in the **WITH** statement.

For example,

```
WITH  x.y^ DO
   BEGIN
        t := s;
   END;
```

could be interpreted as

```
x.y^t := x.y^s;      OR

x.y^t := s;          OR

t := x.y^s;          OR

t := s;
```

The choice of which of the statements is correct is dependent on the declaration of "y". This translation becomes even more complicated when **WITH** statements are nested.

Difficulties with variable identification similar to Pascal's **WITH** statements are found as well in other languages. Examples include FORTRAN's **COMMON** statement and other languages synonym statements. Resolving naming conflicts is then the responsibility of the translator implementor.

The relation language allows all implementation details of algorithms and all data structure implementations to be hidden. Although all algorithm details are hidden, sufficient information still exists to allow validation of certain software engineering methodologies.

# IV. DESCRIPTION OF SOFTWARE METRIC ANALYZER

A software quality metric analyzer which takes as input source code and produces several software metrics, has been developed for use in our research [HENS 85]. The relation language described in the previous section has been successfully used as a tool to express the intermediate form of the source code. This intermediate form is then translated into a set of relations which are then interpreted to produce metrics. The software quality metric analyzer produces three code metrics, three structure metrics, and several hybrid metrics. This 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 remainder of this section describes the details of the implementation of the software quality metric analyzer. For purposes of discussion, the analyzer is divided into three passes. The first pass depicted in Figure 4 concentrates on the relation language translator. Passes two and three shown in Figure 5 combine to form the actual analyzer which produces the software metrics.

## Pass 1

The pass one which we have developed uses the UNIX tool YACC. Pass one has as input the BNF grammar for the source language to be analyzed, the semantic routines which dictate processing for each production in the grammar, and the source code to be analyzed. A file containing the intrinsic functions peculiar to the source language is also input. The source code to be analyzed is assumed to be syntactically correct. Output from pass one is first, a file containing the code metrics for each procedure; length, McCabe's Cyclomatic Complexity [MCCT 76], and Halstead's Software Science indicators, N, V, and E [HALM 77]. These code metrics are produced in pass one since this is the only pass which has the actual source code necessary to generate these metrics. The second file output from pass one contains the relation language equivalent encoded source code.

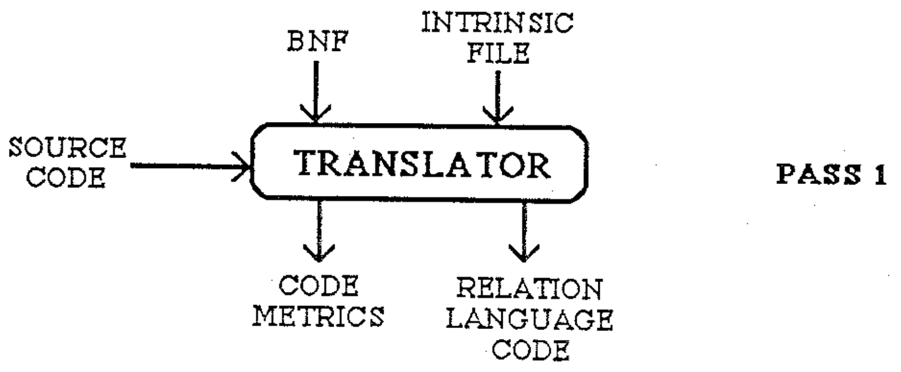

**Figure 4: Relation Language Translator**

## PASS 2

Pass two uses the UNIX tools LEX and YACC. The encoded source code is translated into a "set of relations" which correspond to the source code. This set of relations is completely independent of the original source language. Source code can be processed a procedure at a time. Only the information necessary to generate the relations for a procedure is the encoded source code for the procedure itself.

## PASS 3

Pass three and the associated implementations of the structure metrics are written in standard Pascal. The relation file from pass two generates the three structure metrics, Henry and Kafura's information flow metric [HENS 81a], McClure's invocation metric [MCCC 78], and Woodfield's review complexity [WOOS 80]. The structure metrics together with the code metrics, file one from pass one, produce the hybrid metrics.

As previously stated, pass two is written completely in standard Pascal and is independent of a UNIX environment. The user is in complete control of the selection of metrics to be run and the method of viewing the metrics. This pass is completely menu driven. The user then decides which of the structure metrics he desires to apply to his system. In addition to running the structure metrics and examining the metrics, the user is allowed to define modules, a related collection of procedures, and 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, all related modules as a single level. This feature is especially useful for very large systems. Hardcopies of all reports are available at all times.

Pass one is the only language dependent portion of the analyzer. Previous metric experiments have mandated that we develop the relation language translator. The intent of this paper is to encourage software development organizations to construct a relation language translator specific to their source language, encode some software systems in the relation language, and provide this encoded source code as a testbed for software engineering researchers.

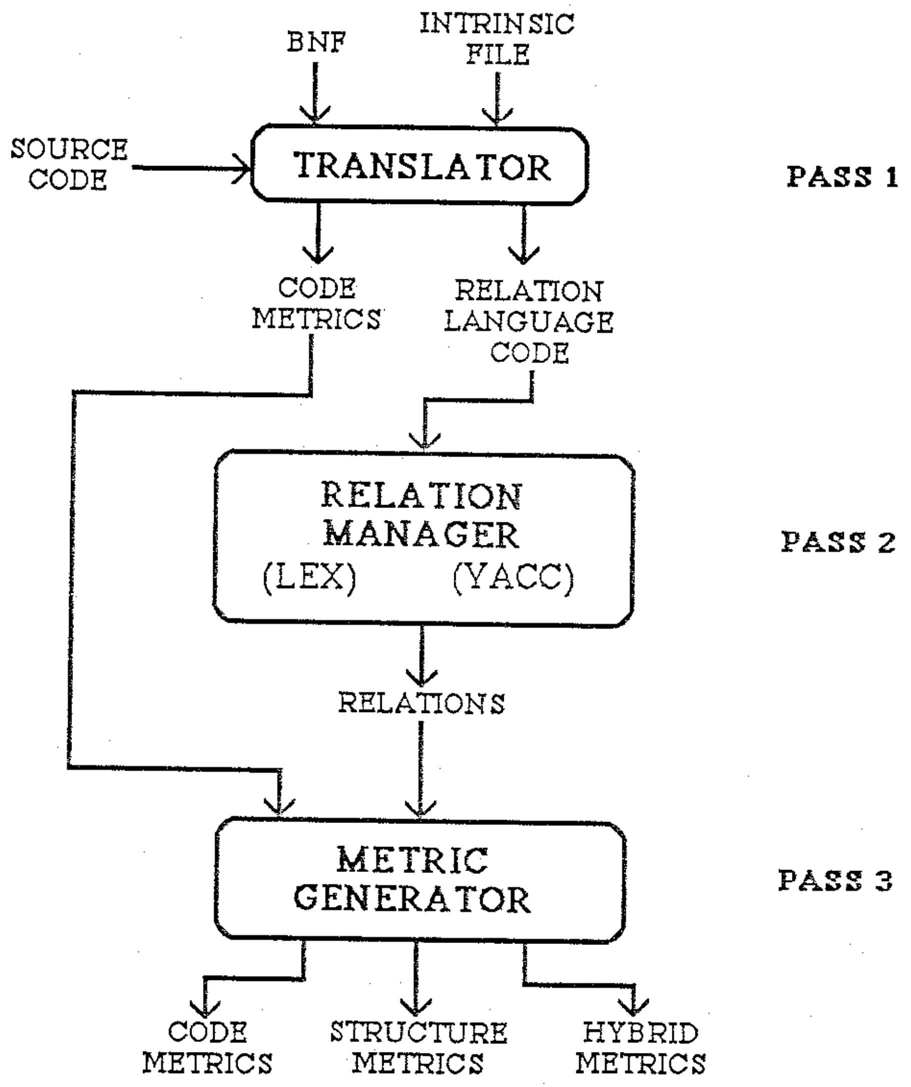

Figure 5: Metric Analyzer

## V. CONCLUSIONS

Have you decoded the algorithm in Figure 1?

If not, focus your attention on Figure 6. This algorithm is written in Pascal with variable names in our simplistic encryption form. Figure 7 displays the same algorithm for the "well-known" **FACTORIAL** function in Pascal.

We believe that the relation language coupled with some form of encryption succeeds in our goal of completely hiding the implementation details of both the algorithms and the data structures in a proprietary software system yet, furnishing sufficient information for the validation of some software engineering tools and techniques.

If even a few software development organizations would provide their source code in a condensed form using the relation language, software engineering researchers would then have a data base of source data for validation purposes

We are not claiming that this language is the answer to all data collection problems, but we believe it will assist researchers with at least some existing data collection problems.

```
FUNCTION F (FVAR1:INTEGER):INTEGER;

   VAR FVAR2,FVAR3: INTEGER;
   BEGIN
      FVAR2 := FVAR1;
      IF (FVAR2 < 0)
      THEN BEGIN  WRITELN ('ERROR FVAR2 < 0')
         ELSE IF (FVAR2 <= 1) /* FVAR2 = 0 OR FVAR2 = 1 */
               THEN F := FVAR2;
               ELSE BEGIN
                          FVAR3 := 1;
                         WHILE (FVAR2 > 1) DO BEGIN
                               FVAR3 := FVAR3 * FVAR2;
                               FVAR2 := FVAR2 - 1;
                         END;
                         F := FVAR3;
                   END;
          END;
   END;
```

Figure 6. Example coded in Pascal with simple encryption

```
FUNCTION FACT (NUM:INTEGER):INTEGER;

    VAR X,TEMP: INTEGER;
     BEGIN
       X := NUM;
        IF (X < 0)
         THEN BEGIN  WRITELN ('ERROR X < 0')
            ELSE IF (X <= 1) /* X = 0 OR X = 1 */
                   THEN FACT := X;
                   ELSE BEGIN
                              TEMP := 1;
                             WHILE (X > 1) DO BEGIN
                                   TEMP := TEMP * X;
                                    X := X - 1;
                              END;
                              FACT := TEMP;
                        END;
              END;
     END;
```

Figure 7. **FACTORIAL** Function

## REFERENCES

[BASV 83] Basili, V.R., R.W. Selby, and T. - Y. Phillips, "Metric Analysis and Data Validation Across Fortran Projects", IEEE Transactions on Software Engineering, Vol. SE-9., No. 6, pp 652-663, November, 1983.

[BOEB 81] Boehm, B.W., Software Engineering Economics, Prentice Hall, 1981.

[CANJ 85] Canning, James T., The Application of Software Metrics to Large-Scale Systems, Ph.D. Thesis, Computer Science Department, Virginia Polytechnic Institute, April 1985.

[CURB 79] Curtis, B., S. Sheppard and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics", Proceedings: 4th International Conference on Software Engineering, p. 356-360, 1979.

[HALM 77] Halstead, M., Elements of Software Science, Elsevier North-Holland Pub. Co., 1977.

[HAMP 82] Hamer, P.G. and G.D. Frewin, "M.H. Halstead's Software Science -- A Critical Examination", Proceedings: 6th International Conference on Software Engineering, pp. 197-206, September 1982.

[HENS 79] Henry, S.M., Information Flow Metrics for the Evaluation of Operating Systems' Structure, Ph.D. Thesis, Computer Science Department, Iowa State University, Ames, Iowa, 1979.

[HENS 81a] Henry, S.M. and D. Kafura, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, pp. 510-518, September, 1981.

[HENS 81b] Henry, S.M., D. Kafura and K. Harris, "On the Relationships Among Three Software Metrics", Performance Evaluation Review, Vol. 10, No. 1, pp. 81-88, Spring 1981.

[HENS 84] Henry, S.M. and D. Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", Software: Prentice and Experience, Vol 14., No. 6, pp. 561-573, June 1984.

[HENS 85] Henry, S.M. and D. Kafura, "Software Quality Measurement: Recent Experience and New Tools", Proceedings of the Eight Minnowbrook Workshop on Software Performance Evaluation Based on Interconnectivity, pp. 64-78, 1985.

[KAFD 82] Kafura, D. and S.M. Henry, "Software Quality Metrics Based on Interconnectivity", Journal of Systems and Software, Vol. 2, pp. 121-131, 1982.

[KAFD 84] Kafura, D., J. Canning, and G. Reddy, "The Independence of Software Metrics Taken at Different Life-Cycle Stages", Proceedings: Ninth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, November 28, 1984.

[KAFD 85] Kafura, D. and Canning, J., "A Validation of Software Metrics Using Many Metrics and Two Resources", Proceedings: International Conference on Software Engineering, London, England, August 1985.

[LASJ 81] Lassez, J., D. Van dir Knijff and J. Sheppard, "A Critical Examination of Software Science", Journal of Systems and Software, Vol. 2, pp. 105-112, 1981.

[LNYW 81] Lynch, W.C. and J.C. Browne, "Performance Evaluation: A Software Metrics Success Story", in Software Metrics (eds. Perlis et. al.) MIT Press, pp. 171-183, 1981.

[MCCC 78] McClure, C. "A Model for Program Complexity Analysis", Proceedings: 3rd International Conference on Software Engineering, pp. 149-157, May 1978.

[MCCT 76] McCabe, T. "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, December 1976.

[MUSJ 75] Musa, J.D. "A Theory of Software Reliability and Its Application", IEEE Transactions on Software Engineering, Vol. SE-1, Nol. 3, pp. 312-327, 1975.

[NASA 81] National Aeronautics and Space Administration, "Software Engineering Laboratory (SEL) Data Base Organization and User's Guide", Software Engineering Laboratory Series SEL-81-002, September 1981.

[PUTL 78] Putnam, L.H., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem", IEEE Transactions on Software Engineering, pp. 345-361, July 1978.

[REDG 84] Reddy, Gereddy, Analysis of a DataBase Management System Using Software Metrics, M.S. Thesis, Computer Science Department, Virginia Polytechnic Institue, June 1984.

[ROSD 77] Ross, D.T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, pp. 16-34, January 1977.

[SHEV 83] Shen, V.Y. S.D. Conte and H. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Emperical Support", IEEE Transactions on Software Engineering, Vol. SE-9, No. 2, pp. 155-165, March 1983.

[SHEV 85] Shen, V.Y., Yu, T.J., Thebaut, S.M., and Paulsen, L.R., "Identifying Error - Prone Software - An Empirical Study", IEEE Transactions on Software Engineering, pp. 317-323, Vol. SE-11, No. 4, April 1985.

[WOOS 80] Woodfield, S.N., Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors, Ph.D. Thesis, Computer Science Department, Purdue University, W. Lafyayette Indiana, December 1980.

[YAUS 80] Yau, S.S. and J. Collofello, "Some Stability Measures for Software Maintenance", IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, pp. 545-552, November 1980.

[YOUE 79] Yourdan, E. and Constantine, L., Structured Design, Prentice Hall, 1979.