EPM/RT-94/26

Beyond Static Metrics

Pierre N. Robillard

Département de génie électrique
et génie informatique

École Polytechnique de Montréal
Novembre 1994

Dépôt légal, novembre 1993
Bibliothèque nationale du Québec
Bibliothèque nationale du Canada

Pour se procurer une copie de ce document, s'adresser:

Les Éditions de l'École Polytechnique
École Polytechnique de Montréal
Case postale 6079, succ. Centre-ville
Montréal, (Québec) H3C 3A7
Téléphone: (514) 340-4473
Télécopie: (514) 340-3734

Compter 0.10 $ par page et ajouter 3,00 $ pour la couverture, les frais de poste et la manutention. Régler en dollars canadiens par chèque ou mandat-poste au nom de l'École Polytechnique de Montréal.

Nous n'honorerons que les commandes accompagnées d'un paiement, sauf s'il y a eu entente préalable dans le cas d'établissements d'enseignement, de sociétés ou d'organismes canadiens.

Beyond Static Metrics

Pierre-N. Robillard,

Laboratoire de recherche en génie logiciel
Ecole Polytechnique de Montréal
C.P. 6079, Succ. Centreville, Montreal, Qc. Canada, H3C 3A7

Tel. 514-340-4238 Fax 514-340-3240 E-mail robillard@rgl.polymtl.ca

Abstract :

This report proposes three ways to look at information obtained from the static analyzer tool, each corresponding to the level of knowledge processed by the static analyzer: lexical, syntactical and stylistic. The static documentation is the listing of the information resulting from the static analysis. The content of this documentation could be drawn or visualized. Static measures are obtained by counting components of information. Static predictions are based on static measurements. This approach outlines the information content of each measure and enables definition of predictive domain of a measure. Examples taken from industrial projects illustrate typical types of information at each level. New application avenues derived from this model are discussed.

1. Introduction

This report gives an overview of the various uses of data obtained from static analyzers and proposes a new model that integrates the various aspects of static software analysis. The purpose is to outline the information content of this in documenting, visualizing, measuring and predicting software characteristics. This approach will provide the reader with a comprehensive understanding of the state of the art and show the benefits and limitations of this data.

The framework we shall adopt in this report is that a program is a kind of information structure, just as a book is a kind of information structure. We call this approach static analysis because it looks at information structure and not the information content or the application domain of the program. The static analyzer is a tool, which extracts specific

data from the information structure. Three activities are conducted with the results of the static analysis: documentation, measurement and prediction.

Measurements based on static analyzer data have been used for more than 20 years. Few outstanding results, if any, have been published. Failure might be due to our limited understanding of static analysis data and unfounded expectations of their benefits.

An overview of the evolution of static analysis is presented. This review does not pretend to be comprehensive, but rather outlines the evolution of the domain. The static analysis of source code was initially associated with the measurement of program complexity and the word *metric* was introduced. Later, the word *metric* was generalized to mean any measure related to software, i.e. productivity metrics, reliability metrics, etc. Today, measures from static analyzers are only a small component of the software metrics world.

We can identify three phases in the evolution of the domain. The first phase, which lasted 10 years starting in 1975, was dominated by work on static measurements and the so-called complexity metrics. The second phase (spanning the next five years) was characterized by enthusiasm for various metrics. The third phase promoted the implementation of metrics programs. [Dumk93] prepared a comprehensive, subdivided bibliography of most of the works published in software metrics. The following textbook references are indicative of this evolution.

The pioneering work of McCabe [McCa76], which introduces the cyclomatic number, and Halstead [Hals77], which introduced Software Science, have had a strong influence on the perception and usefulness of static measures. These metrics are still in use and are part of metrics implementation programs and research projects.

The following three references are typical of the second phase. They introduce metrics, their applications and some instructions for metrics implementation programs. Conte [Cont86] introduces the reader to the world of software metrics and models, their ultimate goal being to enable better understanding and control of the development process, and better estimation of its cost.

The central point of Zuse's textbook [Zuse91] is to provide readers with the necessary knowledge and tools for critically evaluating existing and future software complexity measures. An additional goal is to prepare the reader for a detailed study of the methods of application of measurement theory, the definition and use of scales, the description of measures as an ordinal or a ratio scale, the methods for analyzing the static complexity of programs, the types of complexity measures used to analyze the static complexity of programs and applications of software complexity measurements in practice. The book presents two ways to discuss software complexity measures, the first a theoretical foundation of the measurement of software complexity, and the second detailed discussion of more than ninety software complexity measures and their application to software complexity measurement.

Fenton [Fent91] in his book describes a rigorous framework for software metrics, the obligations one has regarding measurement, the things one will want and need to measure, and the approaches that attempt to do so. It enables one to determine for oneself whether the published metrics and models are doing the job they claim to do and the one that is

wanted. It also shows what to do if they are not. The book shows how to cut through the myriad of metrics and models that appear in the literature and see how and where these fit into a software engineering measurement framework.

The following references are related to the third phase and are concerned with the implementation of a metrics program.

The purpose of Jones'book [Jone91] is to demonstrate that software is in fact capable of accurate measurements, and that the measurements have notable practical value to both the management and the technical community within the software industry. The content of the books is intended to cover all the broad topics and most of the specific ones associated with starting a full corporate measurement program that encompasses productivity, quality, and human factors.

Brady's book [Brad87] on implementing software metrics programs help one to better develop these programs in one's organization, analyze our approach to software metrics implementation and decide which metrics are the most appropriate for one's development and implementation processes, describe some immediate benefits at the project level and anticipate difficulties.

Moller and Paulish's title [Moll93]: *Software Metrics: A practitioner's guide to improved product development*, shares successful experiences with applying software metrics to project management. It contains practical suggestions for implementing metrics, and also contains summaries of successful techniques on metrics implementation and the benefits which resulted within industrial organizations.

The purpose of Hetzel's book [Hetz93] is to show how to make software measurement work in an organization. The book is also about getting working software practitioners and managers to use measurements routinely and to enthusiastically embrace measurement as a way of life. Finally, this book is about getting all of us to take greater professional responsibility in presenting measurement data.

This paper integrates all these viewpoints into a comprehensive model of the information content resulting from static analysis. Section 2 describes the static analyzer's three levels of processing. Each of the following sections presents the three levels for processing static analysis data and illustrates them with examples: documentation, measurement, and prediction.

The work and ideas presented are based on many years of experience with static analyzers in an industrial environment at BELL CANADA and a research environment at École Polytechnique de Montréal. Projects on which static analysis was conducted cover management information systems, compilers, network management systems and real-time embedded communication systems.

2. Model of static information structure.

The information that can be extracted from a system depends on the knowledge we have in this system. A static analyzer is an information extracting machine. The more knowledge encoded in the static analyzer, the more information can be gleaned from the

analysis. If the only knowledge encoded in an analyzer is the recognition of a carriage return character, then the only information that can be extracted is the identification of carriage return characters or the number of lines.

On the other hand, if we can encode the knowledge needed to recognize a good structure programming style, it is likely that information on structure programming workmanship could be extracted. A problem arises when more information is expected from the static analyzer than the encoded knowledge allows. The model proposes a classification for the level of knowledge encoded in the static analyzer and the various types of information processing resulting from the static analysis.

The static analyzer extracts components of the information structure at each level of the analysis: tokens at the lexical, statements and relationships at the syntactical and workmanship or style at the stylistic level. The next section describes the type of knowledge encoded at each level.

The resulting information can be processed in three ways: listed, counted or correlated. The listing of information is called the program's static documentation. The counting of information components is called static measurement and the use of information content to fit a predictive model is called static prediction.

Each way of processing information is related to the level of knowledge available in the static analyzer. Table 1 shows the relationships between the processing level and the information processing type.

Table 1 Model of information processing type based on knowledge level.

Processing type/ Analysis level	Static documentation	Static measures	Static prediction
lexical	listing	counting tokens	based on vocabulary size
syntactical	drawing	counting relationships	based on relationships
stylistic	visualization	counting model characteristics	based on modeling

3. Static analyzer.

The purpose of static analysis is to extract accurate and comprehensive data from the source code. This extraction is based on encoded knowledge. Three mechanisms are used to encode knowledge: regular expressions, grammar rules and algorithm implementation. This section introduces the three levels of static analyzer processing corresponding to the encoding mechanism: lexical, syntactical and stylistic.

3.1 Lexical analysis

Lexical analysis is the first step in static analysis. The lexical information is based on the tokens composing the information structure. Lexical analyzers use regular expressions to identify tokens. This can be done with a software tool called LEX.

The three basic groups of token are the symbols (+,-,*,/ etc.), the reserved words (IF,DO, FOR, etc.) and others. These last could be subdivided into identifiers, strings and comments, for example. Figure 1 shows typical LEX expressions.

The granularity and the specificity of the lexical analysis depend on the accuracy of the regular expressions. For example, the simplest analysis recognizes only the Carriage Return character to count the LOC (lines of code), while a comprehensive analysis recognizes all the words and symbols composing a program (variables, function names, key words, symbols, etc.). The lexical analyzer defines the vocabulary used in the following levels of analysis. The data obtained from the lexical analysis is programming-language-dependent.

An application improvement is to formally measure the information content of the regular expressions composing the lexical analyzer. This measure can provide a basis for specifying the recognition power of a lexical analyzer or its vocabulary size and accuracy.

```
/* symbols */
        "&&"                                { return AND }
        "("                                 { return PARL }
        ")"                                 { return PARR }
        ";"                                 { return PTCOM }

/* reserved words */
        "while"                                     {return WHILE}

/* others*/
        /* identificators */
                [a-z_A-Z][a-z_A-Z0-9]*       { return
IDENTIFICATOR }
        /* unknown */
                .                           { return ERROR }
```

Figure 1 LEX EXPRESSIONS for a WHILE

3.2 Syntactical analysis

Syntactical analysis uses a set of production rules to identify the link between tokens and

Rules can be written at various levels of complexity. The comprehensive analyzer has a complete set of rules. Context-free programming languages like Pascal and C are easier to analyze than context-dependent language like FORTRAN and COBOL. Figure 2 shows a typical YACC input file composed of BNF expressions. The data obtained from the syntactical analysis is programming-language-dependent.

One application improvement is to formally measure the information content of the production rules composing the analytical analyzer. This measure can provide a basis for specifying the capacity of the syntactical analyzer.

```
/* statements list definition*/
    statements_lst      : statements
                        | statements_lst  PTCOM  statements
                        ;

/* boolean_expression definition */
    boolean_expression  : IDENTIFICATOR
                        | boolean_expression  AND  IDENTIFICATOR
                        ;

/* statements definition*/
    statements  : WHILE PARL boolean_expression PARR statement_lst { /* process information */ }
                ;
```

Figure 2 BNF expression for a WHILE construct.

3.3 Stylistic analysis

Stylistic analysis is derived from a model of the programming constructs and is based on syntactical and lexical analysis. Data on the style of the program information structures is then extracted from the source code. Structured programming is an example of construct programming style. Style detection results from the implementation of an algorithm that defines it. The algorithm captures the knowledge on style.

Research could be done to define a taxonomy of styles based on the types of tokens and syntactical rules they are using. A formal definition of style could also be explored. Style could be related to design maturity. Build-and-fix programs might not have the same programming style as formally code-inspected programs.

A generic and comprehensive model is needed to represent and integrate our knowledge of computer programs. Such a model should include all the tokens and their relationships. This will increase our understanding of programming style and enable us to find relationships between programming style and quality of design, readability, maintenance effort, reliability, programming effort, etc. The quality of the documentation obtained from the static analyzer depends directly on the capacity of the various analysis levels. The more

accurate and complete the model representation is the more useful the static analyzer data will be.

4. STATIC DOCUMENTATION.

Static documentation should not be confused with program documentation, which is related to the content or the application domain of the program. Program documentation usually refers to the internal documentation of the program and is made up of comment statements and significant variable names. Static analyzers could detect the occurrence (quantity) of internal documentation, but not assess its quality content.

Static documentation is useful for identifying and locating the components of a program. Some software is still built using an opportunistic approach or on a trial-and-error basis. Programming functions, constructs, statements and variables are assembled by the programmer to implement a functionality. The information structure is not the result of a well-thought-out design process. Programming tokens and statements are written as needed. Static documentation provides the information structure of the program. Static documentation is useful for understanding programs, whether for maintenance, development or quality control activities like inspection. In an ideal environment where all the tokens are well designed before programming, a static analyzer will still be useful to confirm that the program is written according to the design.

The following show examples of the static documentation resulting from each level of analysis.

4.1 Listing

A useful way to represent the data obtained from lexical analysis is to provide the list of tokens at the system, module or statement level. The system level is made up of comprehensive cross-referenced information between files, modules and functions, and identifies functions with identical names within or across executable modules, listing all the function names within a library. At the module level it provides lists of functions, global variable names, parameters, etc. At the statement level it provides type of data and control structures, comment lines, variable declarations, and much more. It is a powerful tool for checking the completeness and integrity of a software system.

The listed part of the static documentation can be stored in a database and programmers can generate queries to help them understand the software product. Examples of such queries are: list all the functions called by a given main function, list the global variables used in a given function, list all statements that use a given variable name, etc.

4.2 Drawing

Syntactical analysis provides the relationship between the tokens. This information is best represented by drawing tools. Drawing is defined as the technique of representing an object in its parts. Drawing should not be confused with visualization which refers to interpretation, as will be seen later. Sophisticated software tools provide drawings at various levels of abstraction and where all display elements are connected to the source

code. The user can toggle back and forth between the drawn views and the corresponding programming components.

Figure 3 shows a typical call tree. Each box represents a software unit and the lines show the links to related software units. The tool automatically finds the boxes connecting any selected box. Others features are finding recursion paths in function calling sequences and displaying the hierarchy of include files, for example.

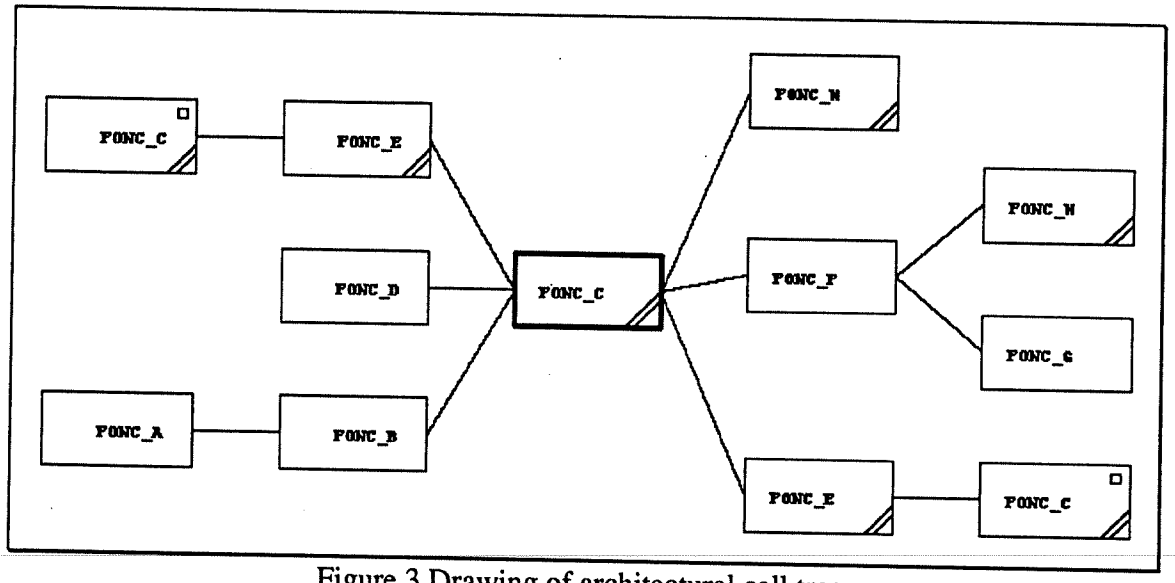

Figure 3 Drawing of architectural call tree

The three-dimensional view in Figure 4 shows a global view of the links between routines. The horizontal axes represent the number of calls made to (x) and from (y), the software units. The vertical axis represents the number of software units. Highly interlocked routines are easily identifiable. Such a representation of measures gives an overview of the system and enables the designer to better understand the coupling of the system and take action on the most needed functions.

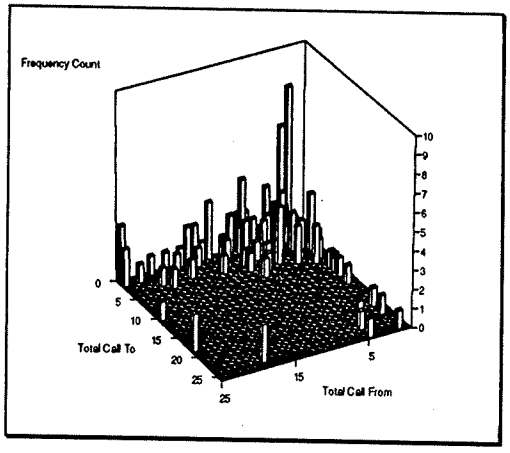

Figure 4 3-D View of the links between software units.

Ideally, any request regarding token location and its relationship to the program could be listed or drawn. Static documentation is an inventory of the information structure of a program.

4.3 Visualization

A style is a particular technique by which something is done, created or performed, or a convention with respect to spelling, punctuation, capitalization and typographic arrangement and display, followed up in writing or printing.

Stylistic analysis is based on a model or a convention. An example of printing style is statement indentation, while structured programming is an example of programming style.

The results of stylistic analysis can be tabulated as, for example, a list of indented statements or a list of breaches of structure. The results can also be visualized according to a model, for example, control flow, data flow, etc. Visualization is defined as the process of interpreting in visual terms or of putting into visible form. In this case, visualization refers to expressing a model in a visual form.

A typical model type is illustrated by the visualization of the control flow. Control flow could be represented as an assembly drawing. In this case, the representation is limited to showing the link between the control statements in a structured program. The simplest visualization mechanisms provide a virtual sketch of the source code, in which case the information cannot be traced back to the source code.

However, formal representation of control flow style enables a link to be made with the source code. The following shows a typical example of formal style definition and its automatic representation. It is called iconic control flow visualization [Robi93].

Programs are decomposed into basic blocks. Each basic block is represented by an icon. Icons are assembled according to defined rules. The block definitions, the aggregations and association rules define the conceptual model. The process keeps all control-related information as it appears in the source code. The icons provide a visual representation, which is independent of the size or complexity of the control flow. The representation is also programming-language-independent.

A basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halting, or the possibility of branching, except at the end. The representation of the relationships among the basic blocks forms the control flow graph.

A basic block is represented by a vertical line with an entry and an exit connector. There are four kinds of connectors: sequential, terminal, label and branching. A node, represented by a horizontal line, links connectors to programming language statements.

The following describes the various types of connectors applied to the basic blocks, and the resulting icons. Sequential connectors are represented by empty circles. Figure 5A shows a basic block terminated by sequential connectors. Figures 5B and 5C show blocks with terminal connectors that define the beginning or the ending program nodes. A label

connector is represented by an arrow-tail at the beginning of a basic block. Figure 5D shows a basic block inputted by a label and outputted by a sequential connector. A branching connector is represented by an arrow-head at the end of the block. Figure 5E shows a sequential block with a branching end connector.

A node is needed when block ends have more than one connector: for example, a block which can be entered by a sequential or a label connector, or a block which can be exited by a sequential or a branching or multibranching connector. Figure 5F shows a block which can be entered by a label or a sequential flow. Figure 5G shows a block which contains a conditional statement, so that it is terminated by two branchings.

A backward branching is a basic block that reverses the normal top-down sequential flow. Figure 5H shows the representation of a backward branching block with a sequential node and a double vertical line terminated by a branching. Formal aggregations and association rules are used to group the resulting icons of a program together to represent the complete control flow.

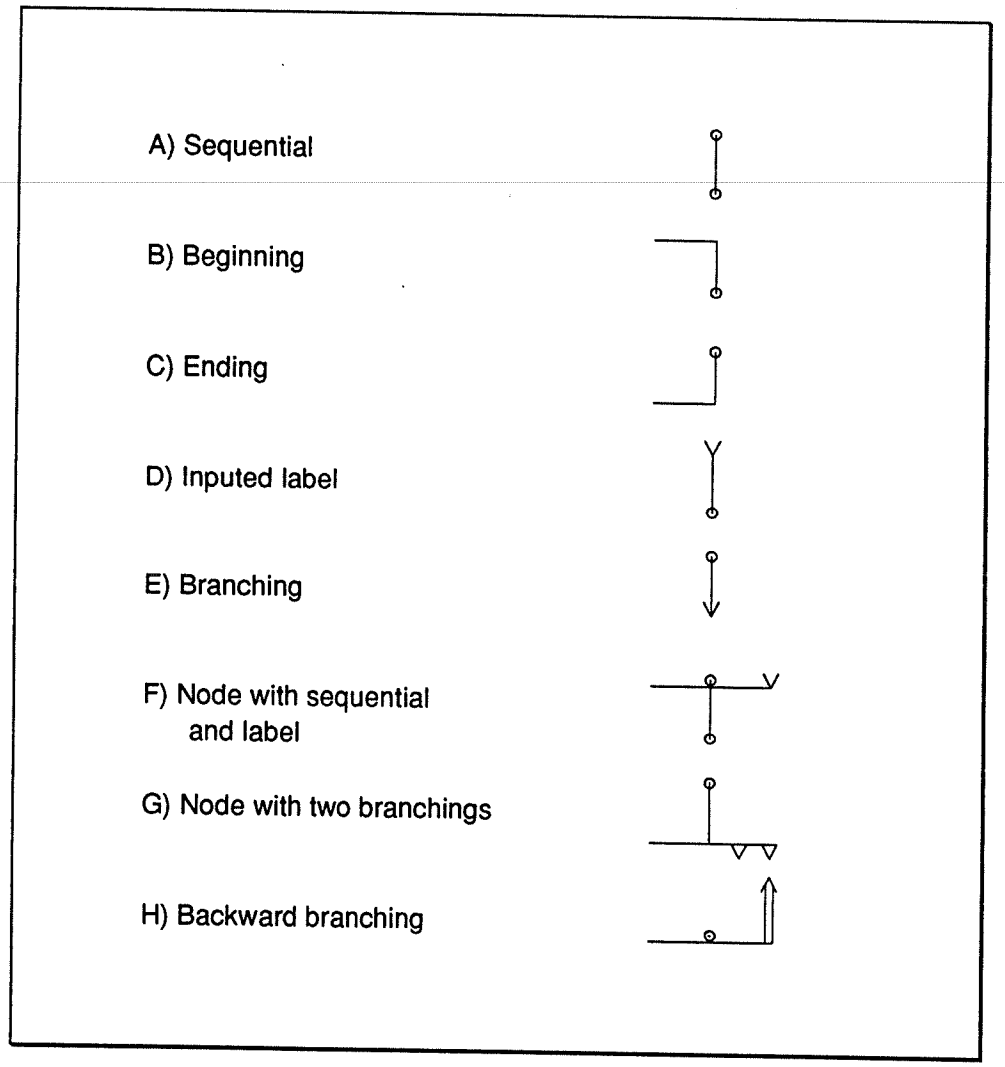

Figure 5 Basic block icons

Figure 6 illustrates the resulting control-flow representation. Directed arcs link origin nodes to destination nodes. A forward arc follows the normal flow of execution (drawn as a single line). A backward arc follows the reverse flow, as in the return arc of a loop (drawn as a double line). The node number is a pointer to the line statement in the source code. A dynamic visualization can display the source code corresponding to any arc.

Half-plane representation allows the identification of arc crossings. The crossings that violate the rules of structured programming are called breaches of structure, and are shown by square dots.

This graphical representation of the control flow corresponds to the general interpretation of graph theory. Arcs contain statements from the source code. Nodes are the logical states of a program where arcs go from one node to another.

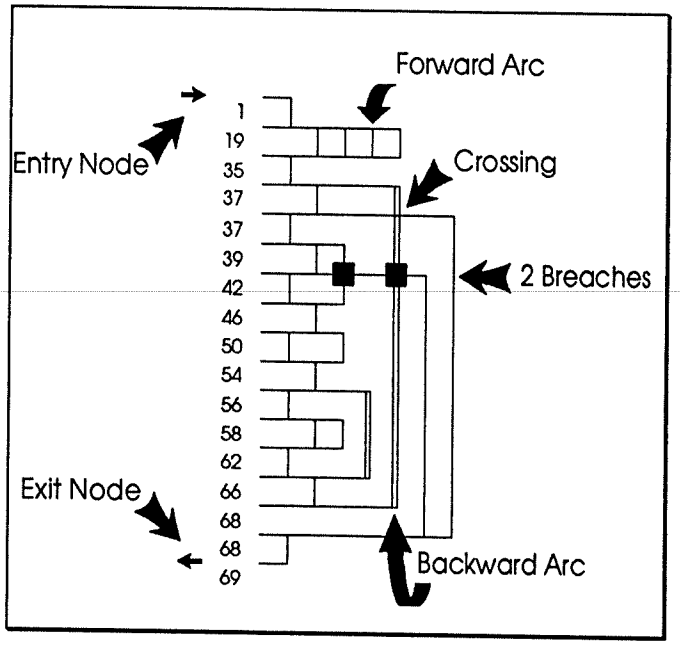

Figure 6 Final representation of the control graph.

Stylistic analysis is based on defined styles. The style could be defined at the lexical level by requiring special wording for variable or function names or at the syntactical level by requiring, for example, that all IF constructs have an ELSE part. It could also be on a formal model, as shown for the control flow. Visualization of the stylistic analysis enables the programmer to see its conformance to the style.

5. Static Software Measurements

Static metrics are defined as measurements made from static analyzer data. They are three types of measurements corresponding to the type of analysis: lexical, syntactical and stylistic.

The meaning of the terms *metrics* and *measure* have been discussed in the literature [Zuse91] [Fent91]. We use the term measure, which is defined as follows:

> *A measure is an empirical objective assignment of a number (or Symbol) to an entity to characterize a specific attribute.*

This section describes and illustrates with typical examples the type of measurements that can be done at each level of analysis.

5.1 Counting Tokens

Lexical analysis provides the list of programming language tokens, and counting these tokens constitues the measurements from the list. They are on absolute scale. Number of operators, number of operands, number of IF key words, etc. are examples of well-known measures of this type. Software Science [Hals77] is based on such measurements. Some measures may have dubious meaning. Exhaustive lists of these measures are available in the literature [Coté] [Zuse91].

Statistical analyses have revealed not surprisingly, that most of these lists of token-type measures are correlated to the size of the software unit [Robi91]. The larger the software unit, the more tokens they have. None of the measure distributions shows statistical normality, which makes conventional interpretation of average, mean and other moments almost meaningless. Figure 7 shows the project distribution for the number of lines of code (LOC). LOC are detected by the occurrence of the CR tokens as discussed before.

On the horizontal axes are the functions in decreasing order of measurement values from the left-hand side. On the vertical axis are the measurement values for each function; the distribution shows the full range of measurement values for the project. This distribution is typical of any item in the list of tokens.

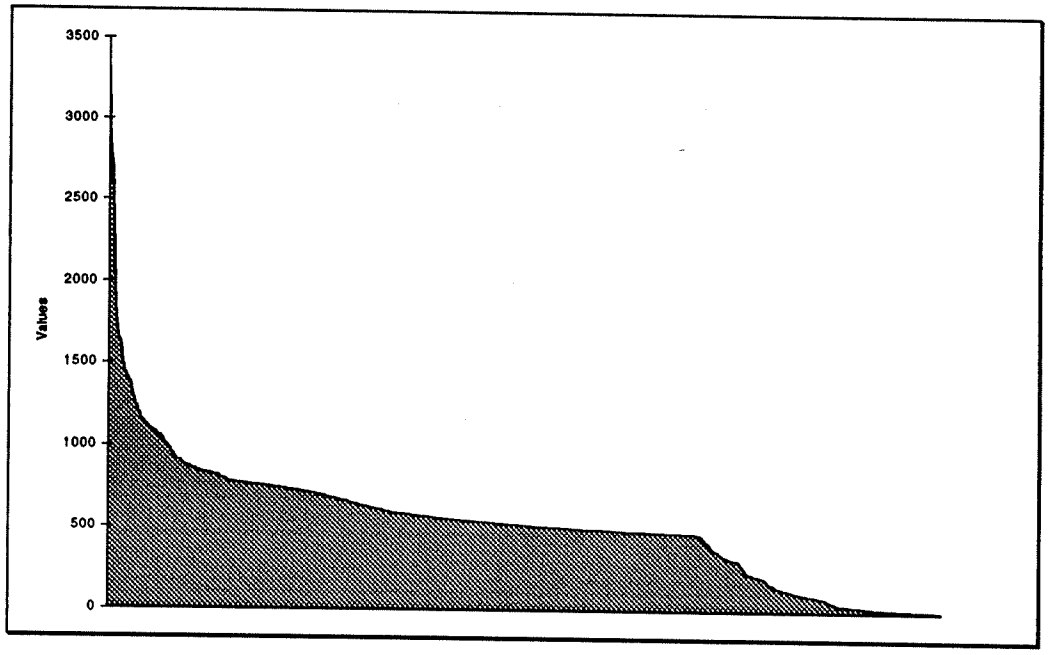

Figure 7 Distribution of the number of LOC

Such measures are useful in evaluating the relative and absolute sizes of a software unit within a system. Development practices could varied depending on various size measurements.

5.2 Counting relationships

Measures based on syntactical analysis take into account the relationships between the tokens. Such measures are: conditional span, unreachable statements (dead code), fan-in, fan-out, nested levels. These are counting measures and are evaluated on an absolute scale. Some of these measures are indicators of structure problems or bad implementation. Dead code is an example of this. Figure 8 shows the distribution of the maximum of nested levels per software unit. These measures could be used to identify inspection schedule priorities among modules.

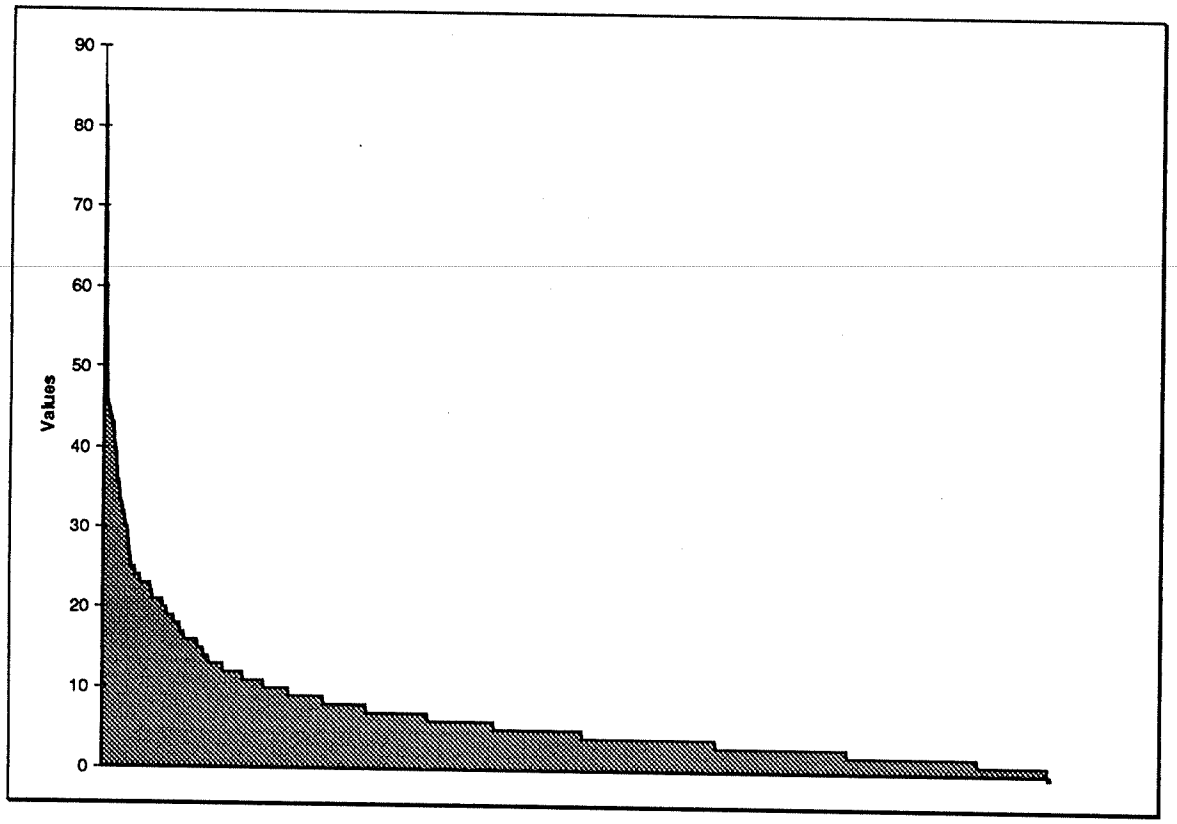

Figure 8 Distribution of the maximum nested levels

5.3 Counting Model Characteristics

Measures based on style analysis are derived from a model of the components of the software product. These measures could be of the ratio or absolute scale type. Breach of structure, cyclomatic number, the measure of interconnectivity and the number of paths are examples of this.

Figure 9 shows the project distribution for the number of paths per module. On the horizontal axis are the functions in decreasing order of the measured value from the left-hand side. On the vertical axis are the measured values for each function. The distributions show the full range of the measured values for the project. Testing policy can defined the usual range of values for a project and the identification of any functions that have out-of-range values. In this example, functions with more than 10,000 paths are identified as being out of range.

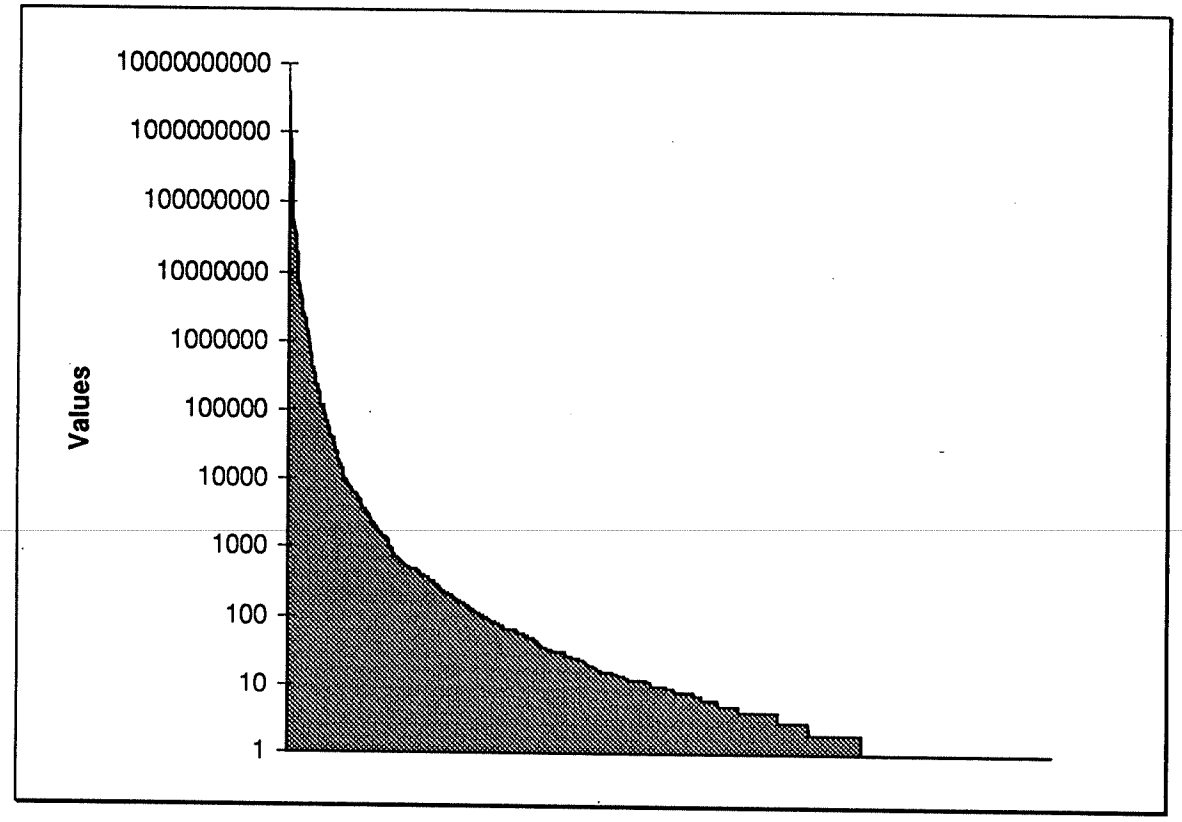

Figure 9 Number-of-paths distribution.

5.4 Measurements representations

Most of the measures could be represented by a distribution. The usual range is determined for each measure distribution. A percentile profile for a project shows the number of functions that fall within the selected range of measure values. All the functions that have unusual measure values are identified and listed.

Figure 10 shows the percentile profiles for 15 selected measures. Percentile profiles consist of two-color columns. The black in each column represents the percentage of unusual functions. These are below or above the usual range of values. Black in the upper part corresponds to the percentage of functions exceeding the range's upper bound. Black in the lower part corresponds to the percentage of functions below the range's lower bound. The darker the profile, the more unusual the project.

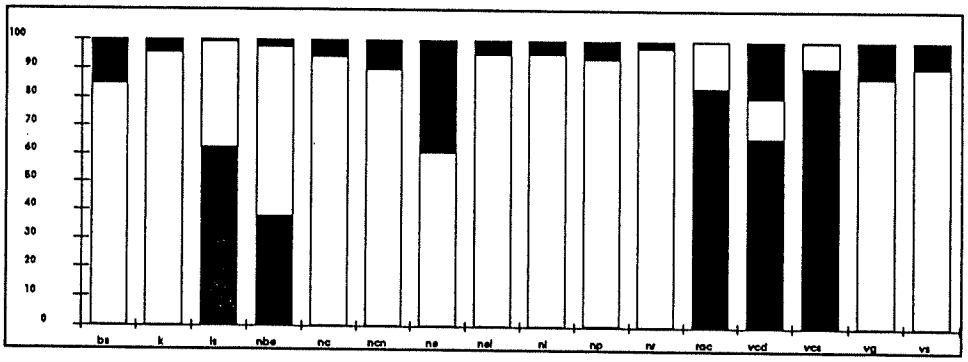

Figure 10 Percentile profile.

This way of handling measurement data is truly objective and does not assume any prediction on the values. It provides the project manager with various measures of the project. The following examples illustrate the benefits of these measures. A measure of the size of the internal documentation (number of comments) shows an under-documented group of modules. The programmers involved in writing these modules were trained to improve their documentation skill. Modules with an unusual number of paths were revised and design inspections were enforced. The measures and their distributions are powerful tools to support project manager decisions.

Usual range is defined by the state of the practice within an organization. Usual measure ranges are actually empirically determined by software engineers and project managers working on a given project. The definition of *usual range* depends on the type of organization, software application and the development environment.

Research is currently in progress to find a statistical approach to the definition of *usual range*. A formal and documented definition will enables organizations to share their data of usual-range behavior and then evaluate project improvement using new practices or processes.

5.5 Interconnectivity

Interconnectivity measures integrate the structural as well as the textual aspects of a program in such a way that the organization of a program can be seen graphically. They integrate the principal attributes of a program, including control flow, data flow and program size into the same model. **[Robi90].**

The interconnections among statements are represented in a table that integrates data flow, control flow and program size. Rows represent statements, and columns represent variable definitions, variable redefinitions and control structures. The level of interconnection among statements within a program is calculated using the information theory concepts of entropy and excess entropy.

We view a program as a system that is divided into many subsystems, in hierarchical order. This approach dovetails perfectly with the structured programming concepts, where each component has one input and one output, and may also be divided hierarchically. Two

types of possible interconnection among statements are identified, those due to the various execution sequences and defined by conditional statements, and those generated among statements themselves by the definition or redefinition of variables.

A table comprising statements, variables and structures is used to represent a program and its attributes, control flow data flow and program size. In this representation all the attributes are integrated to produce a coherent evaluation by the measures. The statements correspond to the rows and the structures to the columns. The table is filled with interconnectivity data.. The data flow corresponds to the links between statements that the variables produce. The control flow is determined by the links between statements occurring in the same control structures. The elements in the interconnectivity table are zeros or ones. Ones in a column mean interconnections among those statements. Criteria for filling the table are based on the mathematical model. The value of the interconnections is obtained by analysis of the patterns of zeros and ones.

The degree of relationship among statements depends on the degree of similarity (or difference) in the patterns. The entropy function is used to measure the variability of the patterns. The higher the entropy, the greater the variability in patterns.

The model has been tested with real data to demonstrate its consistency when changes in the program text or structure are made. Structuring rules have been studied and selected based on their capacity to be extracted automatically from the program text according to the attributes considered by the interconnectibility metric. The rules are grouped under four major headings: modularity, simplicity, control structures and data representation.

For every rules expresessed, we make two versions of the same program. Version 1 does not implement the rules whereas version 2 does it. All together, we have 12 different programs with two versions each. Every pair of programs is designed to evaluate the effect on the measure of one, and only one, given rule. The results reveal that the measure is sensitive to every modification. Moreover the measure is sensitive in a positive way and improvement in the program lowers the measure.

Typical inteconnection profiles are shown in Figures 11 and 12. These two programs have exactly the same functionality, but are based on different implementations. The first is a program composed of large nested control structures (more that 40 statements within a structure) with small data arrays, while the second program is mostly written sequentially and has structures that mainly contain fewer than 15 statements. The interconnection profile has proven useful in comparing various implementation strategies.

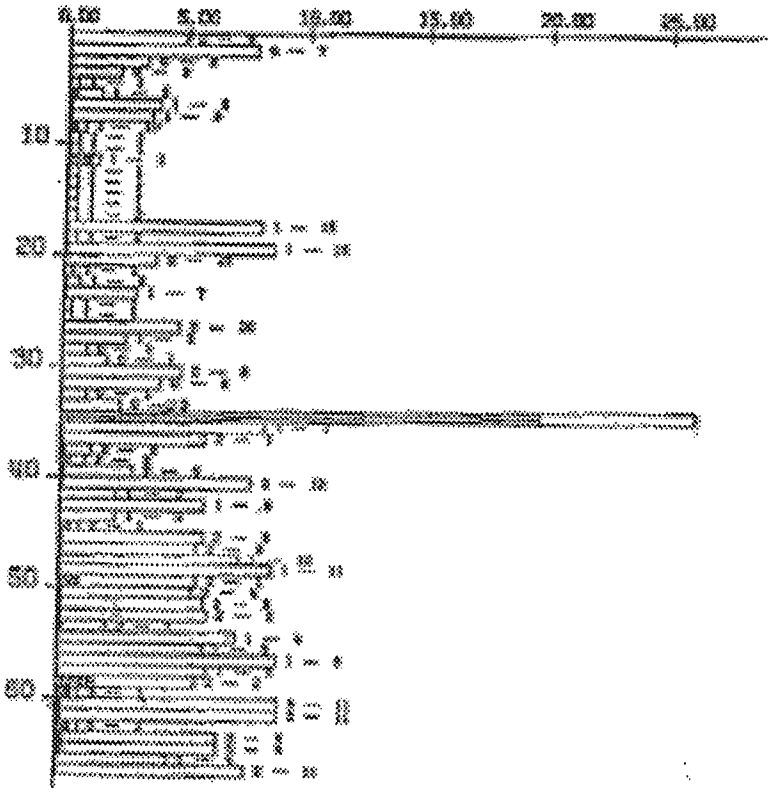

Figure 11 Example of interconnection profiles

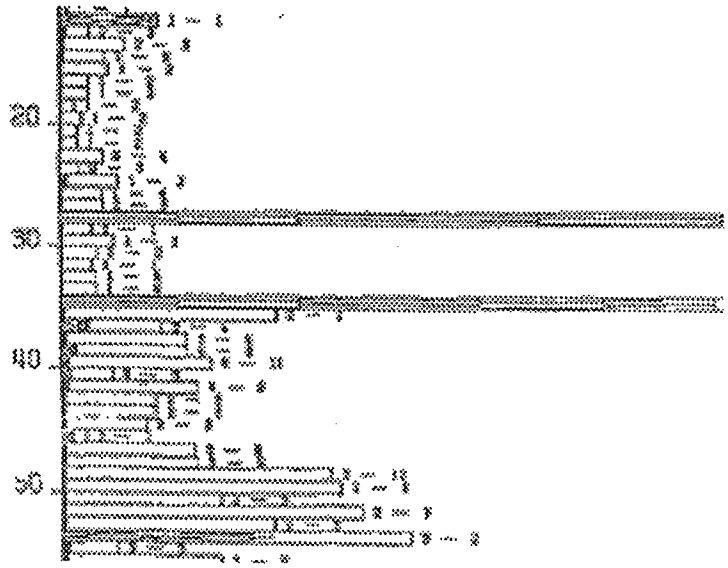

Figure 12 continued.

6. Static Prediction

This section discusses the predictive potential of these measurements. To predict is to forecast on the basis of observation, experience or scientific reason. We called this static prediction because the basis of observation is static measurements.

A measure is an assessment of some entity which already exists. Predicting from a measure is a complex activity that requires the validation of models based on extensive experimental work and statistical analysis [Robi91] [Muns89]. A prediction is an evaluation of an attribute of some entity which does not yet exist. For example, based on the static measure of the software product, we would like to predict its reliability, quality, testability, maintainability, etc.

To do this we need a model that defines an association between static measures and a prediction system [Litt88]. A prediction system consists of a mathematical model together with a set of prediction procedures for determining unknown parameters, and interpreting results.

Prediction is often an association between an internal and an external software attribute. Although predictions among internal software attributes are possible, their usefulness is questionable.

6.1 Based on tokens

The information content of the measure resulting from counting tokens is limited and is strongly size-related. Prediction theories, quality model or practices based on metrics derived from the number of tokens in a program have a very weak basis.

The COCOMO model [Boeh.81] is an interesting example where a lexical measure, number of lines of code (LOC), is used to predict development effort and to estimate time. In this particular case the domain of prediction (effort-time) is not related to the measurement domain (LOC). A successful prediction is accidental and the model has been shown to be unreliable in most circumstances. An analogy with the making of a book will illustrate this point.

The measure of the number of words or sentences could be used as a predictor of the effort or the time needed to translate the book from French to English, for example. However, it is not a good predictor of the effort needed to write it. The effort needed to write a book depends on the subject (application), the experience of the writer in the application domain and the writing activity, and on the tools used: typewriter, word processor, software speller, thesaurus, etc. COCOMO extends the prediction domain of an objective predictor (LOC) by empirically weighting it with subjective factors.

Based on this analogy, LOC is a good predictor of the effort needed to translate good formal design into programming language statements. Any model based on LOC and exceeding this range is more speculative than predictive.

6.2 Based on relationships

Predictions based on syntactical measures are related to the organization of the system.

Fan-in and fan-out are measures derived from the interrelationships of functions, modules and software units. Static observations could be coupled with experience observations such as error rate or maintenance effort, to predict future behavior.

It is important to realize that prediction based on measures of the relationship should be related to the prediction relationships problem. For example, fan-in and fan-out could be used to predict errors resulting from the misuse of global variables or erroneous calling sequences. These measures could be used to predict the effort needed to maintain part of the architecture of the system. Using fan-in and fan-out to predict overall effort maintenance may not be appropriate since some of the effort might be required to modify the internal control structure, data structure and variable initiation which have nothing to do with the basis of the observations.

6.3 Based on models

A model is a description or analogy used to help visualize something that cannot be directly observed.

The cyclomatic number [McCa76] is an example of a simple model represented by a mathematical formula. This model describes the relationship between the node and the edge of a graph made by the conditional statements of the programs. The way to organize the various conditional statements is linked to the programming style. Such a style could be dependent on the application, the design or the programmer'experience. Limiting a module to a V(g) of less than 12 is a stylistic constraint. It is like requiring that there be fewer than 12 sentences in a paragraph. Based, for example, on the premise that smaller paragraphs are easier to understand than larger ones.

Programming styles have various components. Most of the syntactical rules could be used to define stylistic components. Global variables, data structures, pointers, macros, functions calls, etc. are all examples of various stylistic components that could have a strong effect on maintainability, readability, testability etc.

The predictive power of a model is limited by the domain of the observed values. It could be hazardous, for example, to predict program complexity based only on an elementary model of programming construct style.

Predictions based on the stylistic measure could be related to design, maintainability, readability, etc., but may be counter-productive in terms of efficiency.

The measure of the number of paths is an indicator of the various ways the parts could be linked together. Intuitivly we know that the greater this number the more effort is needed to understand or modify the product. This example shows the use of measure distribution to define the profile of a project and identify functions that are unusual according to this profile.

6.4 Based on composite measures

Measures could be grouped together based on their predictive domain. A model is based on the mixing of the various metrics, but care is taken to respect the predictive domain of each measure. In other words, measurements derived from the lexical domain are used to

predict size component only. Measurements derived from the syntactical domain are used to predict organizational behavior, and measurements derived from style considerations are used to predict the style charateristic of a project. The model uses a tree structure where each branch of the tree is a specific predictive domain.

The following illustrates one example of a prediction model based on the three types of static software measures (size, organization and style). The example illustrates the model for predicting the analyzability of a software unit. This is the level of effort needed to understand a software unit during, for example, a maintenance task.

Experiments are needed in order to better understand the association of the various predictive domains and the size of the resulting predictive domain.

6.5 Predicting Software Analyzability.

This section describes the use of static measures to determine the analyzability of a software unit. The analyzability of software modules varies widely within a project. The simplest module might be quickly understood by someone unfamiliar with the application, while others are so complex that even experienced software engineers can spend a significant amount of time trying to understand them.

Experts use a rule-based system to define the analyzability level of a module prior to its modification. The measurement is also taken after modification to evaluate the 'complexity gain' of the process.

We identify five analyzability levels. In practice, there are no clear-cut boundaries between these levels, but rather they form a continuum from one level to another. The following defines the typical modules in increasing order of complexity:

Level 1: Basic Utility Modules. These are the simplest modules. They are usually small and have very few conditional constructs. Their tasks are simple and self-evident. They do not call many modules and are at the bottom of the call graph. The number of paths is very small.

Level 2: Specific Subtask Modules. Subtask modules are more difficult to understand than basic utility modules. They use some conditional constructs and sometimes call other modules. The number of paths is limited, but nontrivial. These modules are often refinements of more important tasks.

Level 3: Switching Modules. Switching modules use many conditional constructs in an organized way. The organization is mostly sequential or nested, but rarely mixed. The number of paths could be considerable. These modules are used to select from among many tasks based on some control variable or calculated condition. The selected modules could be of any level of complexity.

Level 4: Decisional Modules. Decisional modules use many conditional constructs in a mixed way. The task implemented is often part of a more complex algorithm. The goal is usually to compute data based on numerous Boolean expressions.

Level 5: Algorithmic Modules. Algorithmic modules are the more difficult to understand. There are many mixed constructs: sequential, nested, conditional and looped. The number of paths is large.

Modules are automatically associated with an appropriate analyzability level by a classification table based on four measures.

NCN	Number of conditional constructs.	Size
TCT	Total number of calls to other modules.	Syntactical architecture
NL	Number of looping constructs.	Syntactical statements
NP	Number of independent paths.	Stylistic

A team of experienced software engineers has defined and validated the following classification table:

LEVELS	NCN	NL	NP	TCT
1	0 - 1	*	*	0 - 1
2.1	0 - 1	*	*	GT. 1
2.2	2 - 5	0 - 2	201 - 400	*
3.1	2 - 5	0 - 2	1 - 200	*
3.2	GT. 5	0	LT. $0.2NCN^2$	*
3.3	GT. 5	0	GT. $0.8\ NCN^2$	*
3.4	GT.5	1 - 3	1 - 400	*
4.1	2 - 5	1 - 2	GT. 400	*
4.2	2 - 5	GT. 2	1 - 500	*
4.3	GT. 5	0	$(0.2 - 0.8)\ NCN^2$	*
4.4	GT. 5	1 - 3	401 - 1000	*
4.5	GT. 5	GT. 3	1 - 1000	*
5.1	2-5	GT. 2	GT. 500	*
5.2	GT. 5	1 - 3	GT. 1000	*
5.3	GT. 5	GT. 3	GT. 1000	*

Table 6.5.1. Metrics classification table for the 5 levels

The following table presents preliminary results obtained from the classification table. The level 5 modules represent only 17.9% of the modules, but they received 58.7% of the modifications

Level	5	4	3	2	1	Total
Number of modules	117	45	332	65	93	652
Percentage of modules	17.9%	6.9%	50.9%	10.0%	14.3%	100%
Number of modifications	64	3	37	2	3	109
Percentage of modification	58.7%	2.8%	33.9%	1.8%	2.8%	100%

Table 6.5.2. Preliminary results obtained from the classification table.

The data illustrate the following points:

- The module's analyzability levels taken from each project have been validated by groups of software engineers. 5 modules from each level (a total of 25) were given to a team of software developers. Team members had to rate each module according to the narrative module's analyzability definition. Metric values were not available to them.

- Module analyzability is strongly correlated to the maintenance effort for existing projects and to the development effort for new projects.

These results are helpful in planning the effort required for the maintenance of modules. This information is also useful during the implementation phase, since programmers are now aware of the type of module on which they are working. Inspections and walk-throughs could be planned and focused on the type of module analyzability level.

Figure 13 illustrates a typical level distribution. The Y axis contains the number of software units per level. With such a graph, re-engineering, redesigning and maintenance efforts can be more precisely evaluated.

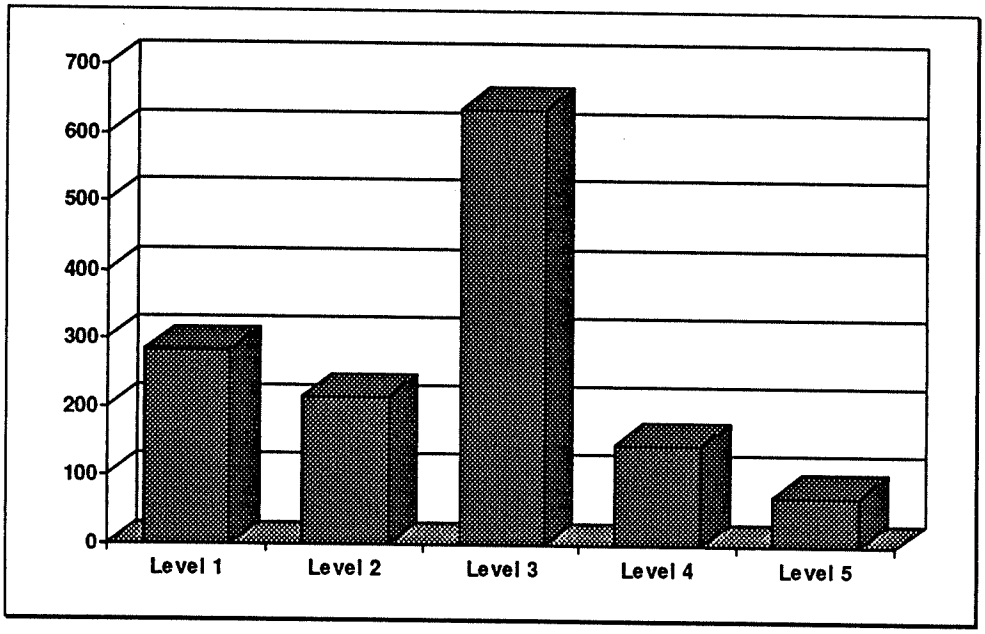

Figure 13 Analyzability level classifications

Controlled experiments are needed to validate the relationship between the metrics' values and the quality factors. A well-designed quality and measurement program can provide indicators for process improvement

7. Conclusion

The use of static analyzers in software engineering is a reverse-engineering approach. The static analyzer extracts the list of tokens (lexical analysis), derives the relationship between the tokens (syntactical analysis) and computes conformance of particular relationships to a defined style.

Each analysis level performed by the static analyzer is based on some knowledge. The knowledge is encoded as regular expressions, grammatical rules or procedural algorithms. This knowledge is used by the software tool to extract information components from the source code. These information components can be used in three ways: they can be listed, measured or used to predict software characteristics.

This approach provides insight on static software measures. Based on this model, we propose new directions in the software application of static measures and in software metrics research.

The static analyzer should be a standard tool in any software environment. Developers and maintainers will benefit from a reverse-engineering and documentation tool such as this. However, it is important to realize that static analyzers are not just front-end compiler. The main difference lies in their target output. Compilers are built to produce efficient machine code and they usually do not record information that is relevant to the programmer. Memory addresses are more important than variable names. Static analyzers as described in this paper are designed to extract information useful for a human being.

Research is needed to formally define the knowledge level of each component of a static analyzer and the information content of the resulting analysis.

A generic model is needed to define software programs in terms of their tokens and their relationships. Such a model will enable us to define the type of knowledge required to do any documentation, measurement or prediction task.

Organizations can define the type of static documentation needed before accepting of a module. This documentation information can then be obtained automatically during the acceptance procedure.

Organizations should clearly distinguish between design documents, the drawing of existing software components and visualization of information based on a model.

The appropriate use of measures can provide insight into software projects. Much confusion can also result from the misuse of measures. Measurement data should not be confused with interpretation of the measurement data. Measurement data constitutes objective information, the content of which has to be formally defined.

Organizations should be aware that the meaning of measures is not always clearly defined. Static analyzers which collect the data are not yet standardized, like compilers are for example, and the computation of the measures is often not documented. The lines of code measure (LOC) is a typical example of this. A line of code could be logical, physical or a mixture of both, and computation can be a combination of the following: new lines, test lines, comments, include files, declarative statements, etc.

This paper proposes a model to make a clear distinction between data, a measure and a prediction based on the measure. It also introduces the predictive domain of a measure. It is important to distinguish between the measure and the interpretation of the measure. For example, the number of Boolean expressions is an objective measure, while the complexity number is the interpretation of the measure. An implicit or explicit model assumes that there is a correlation between the number of Boolean expressions and the complexity of the software. Some measure names convey implicit meanings that could be misleading.

Progress in software engineering that enables the sharing of information demands that the process of collecting data be formalized and the measure-naming convention be standardized.

Theoreticians are building models of software programs to help promote an understanding of the relationships among the data provided by static analyzers. Theoretical work on comprehensive models is needed to improve our understanding of measurements. Programmers could readily use the information obtained from static analyzers to complement the documentation of their software systems. Project managers display measurement distributions to identify any unusual software units.

Designers can verify some of their specifications by visualizing the control and data flows. Finally, software quality engineers can make some predictions on the quality or the risk inherent in the system based on software measurement. Re-engineering, reverse-engineering, testing, and maintenance are some of the activities that benefit from static analysis.

The problem of metric validation remains, however, and can be solved by studies of the relationships between the measures and data on error rates, development costs and reliability indicators. Such studies could be viewed as the next step toward the validation of this model. The neural network is a promising new tool for metric validation, and object-oriented approaches offer new applications for static metrics. We believe that the formal modeling of source code is a prerequisite for any breakthrough in the understanding of static metrics.

8. Acknowledgments

This paper summarizes the work resulting from a team effort. Many people have been working on this project in the past five years. My thanks to André Beaucage and Jean Mayrand for their major involvement in the design and development of the software tool DATRIX, and to software engineers Martin Leclerc and Claude Leblanc for the implementation. Thanks, too, to Dr. Germinal Boloix for his fruitful comments and

discussion on the subject and to the many graduate students who participated in this project, specifically Jean Mayrand (Ph.D.), Jean Sebastien Neveu, Philippe Mathieu and Hélène Beneteau de Laprairie.

This work is supported by BELL CANADA. We are grateful to professionals from Francois Coallier's group at Bell Canada for their supports and comments.

9. References

[Boeh81] Boehm B.W. Software Engineering Economics, Prentice Hall, Englewood Cliffs N.J., 1981

[Brad87] Brady R. B. and Caswell D.L. Software Metrics : Establishing a company-wide program, Prentice-Hall Inc. Englewood 1987

[Cont86] Conte et al, Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Comp., 1986,

[Dumk93] R. R. Dumke, Software Metrics, A subdivided bibliography, Research Report IRB-007/92, Otto Von Guericke University of Magdeburg Gemany, July 1993.

[Fent91] Norman E. Software Metrics A rigorous approach. chapman &Hall London 1991.

[Hals77], Halstead, M.H. Elements of Software Science. New York Elsevier North-Holland, 1977)

[Hetz93] Hetzel B. Making software measurement Work, QED Publishing Wellesley, MA, 1993

[Jone91] Jones C., Applied Software Measurement, McGraw-Hill, Inc. 1991.

[Litt88]. Littlewood, B. Forecasting Software Reliability, Software reliability Modelling and Identification, Lecture Notes in computer Science 341, Springer-Verlag, , 141-209, 1988

[McCa76] McCabe, Thomas J., A Complexity Measure, IEEE Trans. on Soft. Eng. Dec. 1976, pp 308-320

[Moll93] Moller K.H and Paulish D.J. Software Metrics A practitioner's guide to improved product development Chapman&Hall 1993

[Muns89] Munson J.C. and Khoshgoftaar, The Dimensionality of Program Complexity,. Proceedings of the 11th International conference on software Engineering, Pittsburg, pp. 245-253, may 1989.

[Robi89] Robillard P.,N., Boloix G., The Interconnectivity Metrics: A New Metric Showing How a program is Organized. The Journal of System and Software 10, 29-39, 1989.

[Robi91] Robillard P. N., Coupal D., Study on the Normality of Metric Distribution, Proceeding of the 3rd Annual Workshop on Software Metrics, Silver falls, Oregon, March 17-19, 1991

[Robi93] Robillard P. N., Simoneau M., "Iconic Control Graph Representation", Software-Practice and Experience, Vol. 23(2), 223-234.

[Zuse91] Horst Zuse, Software Complexity, Measures and Methods, Walter de Gruyter, Berlin 1991.