EPM/RT-94/12

Practical Experiences with Static Metrics

Pierre N. Robillard

Département de génie électrique
et génie informatique

École Polytechnique de Montréal
Mars 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.

Practical Experiences with Static Metrics

Pierre-N. Robillard, Ph.D.

Laboratoire de Recherche en Génie Logiciel
Ecole Polytechnique de Montréal
C.P. 6079, Succ. Centreville,
Montreal, Qc. H3C 3A7
Tel. 514-340-4238 Fax 514-340-3240
E-mail robillard@rgl.polymtl.ca

Abstract

This paper presents the various applications of static metrics. The expected benefits and the difficulties and limitations are described for each application and the research avenues are explored. The applications are static measurement, statistical analysis, architectural visualization, metric representation, quality profile, software analyzability, control flow representation and restructuring. Each approach is defined and described using a typical example taken from an industrial project. This paper for the most part presents an integrated view of the application of static metrics and outlines some promising research avenues.

1. Introduction

The static measurement of source code was initially associated with the measurement of program complexity. The pioneering work of McCabe and Halstead has had a strong influence on the perception and usefulness of static metrics. Static analyzers have evolved and there is a wide spectrum of tools available on the market today, as a result of which source code parsing can be done at various levels of accuracy. The first generation of tools detects key words related to the control flow, such as IF, DO, FOR. The second generation of tools parses every word and provides some information on the data flow or data structures. The upcoming third generation of tools is based on formal representation of the source code and can provide data based on information theory.

This paper gives an overview of the various applications of static metrics, providing the reader with a comprehensive understanding of the state of the art and showing the benefits and limitations of the first- and second-generation tools. All the results presented have been obtained from industrial projects. Any of the practices presented could be readily implemented, which shows that static analysis has more to offer that just complexity measurement. However, not all aspects of metric applications not covered here. Tests and path coverage have intentionally been left aside.

We present metrics as a way to diagnose or evaluate software quality. Past experience with metrics has shown that the successful use of metrics is based on careful consideration of existing measurements and their assessment. Experts need to look at a project and determine the level of measurement needed. Some parts of a project are trivial and do not need to be measured, and others are so complex that static analyses are of the utmost importance, mostly because of the visualization they provide of complexity. Otherwise,

the software would be unduly difficult to understand. Static metrics are helpful in deciding on maintenance tasks, redesigning, re-engineering and restructuring. This overview shows the potential of static software metrics.

The aim of step 1 is to provide a comprehensive static source code measurement set. In step 2 extensive statistical analysis can be performed to extract metric behavior or define a project profile(1). In step 3, experts use these measures and various outputs from display tools to obtain a multi-view picture of the software. This information is provided at various levels of detail. It can be a graphical executive summary or a specific programming statement. In step 4 these data are used to diagnose the program from various points of view: maintenance, re-engineering, establishing a development guide, improving the process and controlling quality.

Static analysis could also be used in conjunction with other tools to improve the software product or its documentation. For example, structural information could be extracted from a program and fed to a restructuring tool which would automatically restructure existing programs and outline any existing schemas. Schemas are used to understand program plans and also make functions much easier to understand.(2)

The examples and results presented in this paper are based on the static analyzer DATRIX™. Datrix is a third-generation tool. The data presented in this paper are essentially based on the capacity of first- and second-generation tools and are not specific to a given tool. However, control flow representation needs a third-generation tool which is specific to Datrix. This tool records more than fifty metrics. The data can be exported to various software tools for analysis, display and environment integration. For example, The powerful statistical package, SAS™, can be used to derive ANOVA, and factor or discriminant analysis of metric values. A spreadsheet (LOTUS 1-2-3™) can be used to display distributions of metric values. Various viewpoints can be studied by selecting the appropriate set of data.(3)

Such studies will target the metrics most needed to meet project objectives. Usually, a dozen or so of the fifty metrics are kept for general analysis, and the number of metrics is often increased when a specific problem needs to be solved.

The approach described in this paper has been tested over the past three years and integrates the content of various papers already published on the subject. These projects are the result of successful collaborations of the following organizations: BELL CANADA INC. the NATIONAL RESEARCH COUNCIL OF CANADA, SCHEMACODE INTERNATIONAL INC. And the SOFTWARE ENGINEERING LABORATORY AT the ÉCOLE POLYTECHNIQUE DE MONTRÉAL.

2. Statistical analysis of static metrics

Practitioners should be aware that basic statistical studies ought to be conducted before any interpretation is done on the data. Practitioners should also know that statistical methods such as factor analysis ought to be carried out on any set of metrics before spending time trying to find the meaning of a particular metric. A great deal of work is irrelevant because it is not statistically sound. Statistical analysis also presents a systematic approach for deriving the information content of metrics. Static source code metrics constitute one of the techniques available for evaluating the various aspects of software attributes. Research in software engineering has shown the importance of software metrics. For the most part, static metrics are derived from source code token counting and graph control flow characteristics. Research carried out on these classic metrics has generated so many metrics that it is difficult to choose among them. Actually there is no consensus on the use of any particular metric. The objective of this section is to discuss the basic statistical analysis needed before attempting any interpretation of the metric results.

Many metrics are used and various operations are performed on metric values. For example, the metric values of a collection of routines associated with a project can be summed up and averaged. This average value can then be used as an indicator of this metric value for the whole project. Also, the average values of various metrics can be mixed together to form an indicator of overall project complexity or quality.

Such operations on metric values assume that their distributions are known and ideally normal. Studies are needed to verify this hypothesis. Since normality is rarely found, transformation techniques must be tested in order to fit metric distributions into normality. Factor analysis could be employed on metrics to indicate the dimensions that are, in fact, measured. We usually find one dominant dimension and one or two weaker dimensions.

The goal of factor analysis is to determine how metrics are correlated and how many different dimensions are measured by a set of variables. Factor analysis is a field of statistical analysis that deals with multidimensional observations. It transposes observations measured in an N-dimensional space into a reduced space in which interpretation is facilitated. Using the factor model with extraction by the principal components method, the first axis represents the maximum variance between observations. The second axis is found under the constraint of maximizing the remaining variance, and similarly forth for the other axes. We call these axes factors. There is no correlation between factors, which means that each axis has a unique information content and is orthogonal to the other axes.

The results of factor analysis show that more than 60% of the variability in the measurement of classical metrics is represented by only one factor in the projects analyzed. This factor has been identified with size because every volume metric has a big projection on it. In fact, few dimensions of complexity are measured since only four factors are extracted per project. In addition, the last two factors contributed less than 10%. It is observed that these factors could be considered a topological indicator since at one end we find the number of statements, while at the other end the metrics related to the nesting and number of knots is projected.

Despite the fact that factor analysis is a linear model, we have found that many of the metrics that have been proposed by software engineers measure the same aspects of software quality. It is shown that size (or volume) is the principal feature measured by these metrics. Considering this, practitioners should be careful to try to find different meanings in metrics that have the same dominant factor. It may be of no value, for example, to work with a set of metrics that all have the same factor, meaning that their variability will nearly always behave in the same way.

The discriminant analysis shows that metrics could be used to extract some characteristics of the programming environment, and allows separation criteria, which are in turn used to reclassify the data. A very good reclassification indicates that the data is well identified by the projects, the programming languages and the programmers. Both parametric and non-parametric methods exist to complete the discriminant analysis. Because the data are usually not distributed according to known distributions, only non-parametric methods can be used. Two methods used are the kernel method, which uses different techniques to estimate the density, and the nearest-points method, which classifies the observations according to their surroundings. The discriminant analysis also shows that it is possible to find signatures to differentiate routines according to the various projects, programmers and programming languages involved.

The goal in the development of new metrics will be to explain aspects of software that are not currently being measured. A subset of independent metrics will be more useful than a number of different metrics that all measure the same aspects of software quality. The use of multidimensional statistical methods may prove very useful in this search for new metrics or in the validation of the information aspect of others.

3. Architectural visualization

Static metrics could be used to derive the calling tree of a software program, fan-in and fan-out could be measured, the link between all the software units could be derived and product integrity could be assessed. Also, the structure of the tree could be characterized: number of branches, modes, scope and span. Although few indirect metrics try to qualify call graph structure, this information is most useful in call graph visualization .

This could be part of the tool, or the information could be exported to other tools for documentation reverse-engineering purpose. The project call graph obtained from DATRIX™ could be sent to the Software through Pictures™ tool (STP) for integration into the existing environment. The STP structured editor can be used to display and access any call-graph module.

The system architecture can be visualized to determine the module links. Three-dimensional graphs (Fig. 1) show a global view of the interaction between routines. The horizontal axes represent the number of calls made to the (X) and from the (Y) software units. The vertical axis represents the number of software units. Highly interlocked routines are easily identifiable. Simulation can be performed to see the impact of architecture modification.

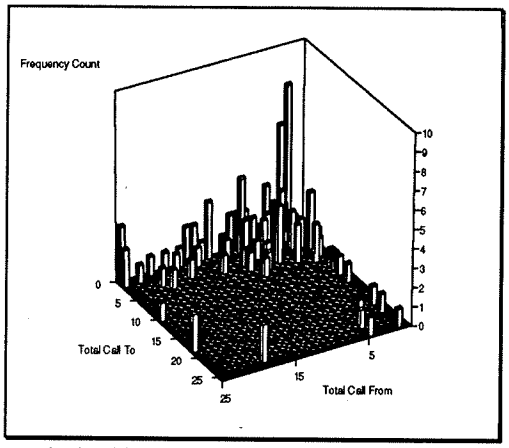

Figure 1. Architectural visualization from *3-D graph.*

Research in this area is needed to find metrics that describe the architecture of the system. Such metrics could be the measure of the organization of the modules or some metric related to entropy. It would be interesting to validate the implemented architecture with the specification or the design. Metrics that predict the maintainability of an architecture would be most helpful.

4. Metrics representation

This section shows examples of metric-based documents used by experts to evaluate software projects. Metric programs need not be extensive, in fact very often, a few metrics can provide helpful information about a project's structural quality. Static metrics provide information for various tasks: modifying system architecture, improving documentation, redesigning modules and integrating a new environment. Experts need to look at project measurement metrics to determine where most of the work is needed and establish priorities within budget constraints.

4.1 Metrics distributions

Commercial spreadsheet software generates metrics distributions based on DATRIX™ measurements. Figure 2 shows the project distribution for the number of paths per module. On the horizontal axis are the functions in decreasing order of the metric's value from the left-hand side. On the vertical axis are the metric's values for each function. The distributions show the full range of the metrics' values for the project. Experts can then work out the usual range of values for a project and identify any functions that have out-of-range values. In this example, functions with more than 10,000 paths are immediate candidates for inspection since they have traditionally proved difficult to understand and test. The usual range is determined for each metric's distribution.

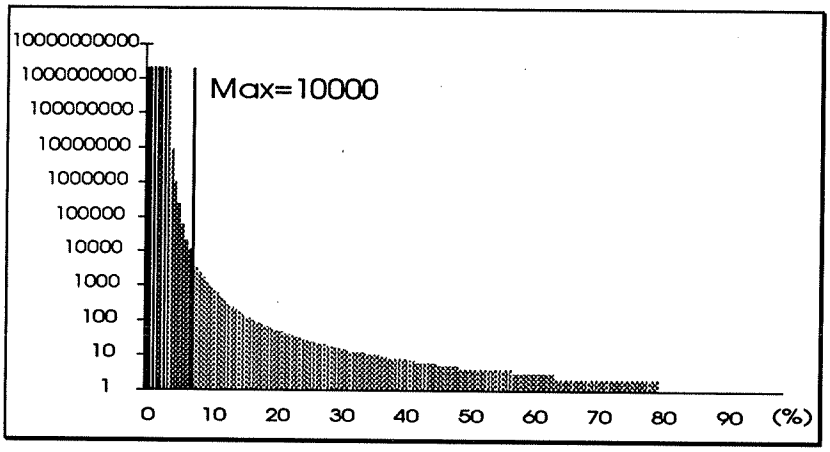

Figure 2. *Number-of-paths distribution.*

4.2 Profile definition

A percentile profile for a project shows the number of functions that fall within the selected range of metric values. All the functions that have unusual metric values are identified and listed.

Figure 3 shows the percentile profiles for 15 selected metrics. 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.

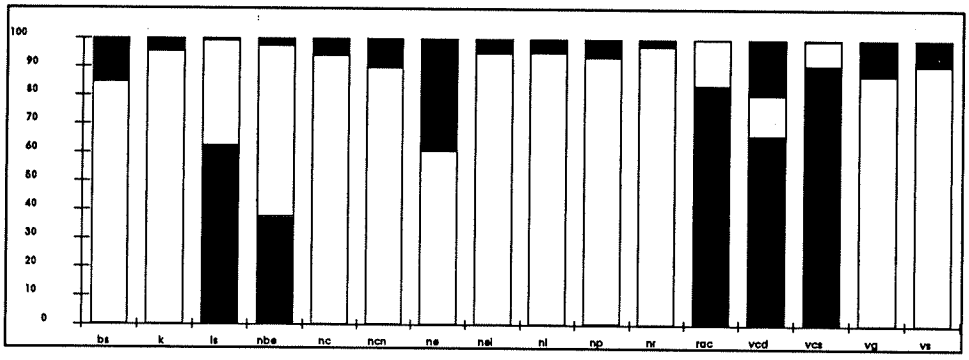

Figure 3. *Percentile profile.*

The grouping of the various metrics may provide information on a quality factor. Typical quality factors could be testability, analyzability, maintainability, portability, correctness, etc. Research and 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.

5. Software analyzability

This section describes the use of static metrics to determine the analyzability of a software module. This example illustrates one of the many applications of software metrics.

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. There are no clear-cut boundaries between these analyzability 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 complex 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 complex ones. There are many mixed constructs: sequential, nested, conditional and looped. The number of paths is large.

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

NCN	Number of conditional constructs.
TCT	Total number of calls to other modules.
NL	Number of looping constructs.
NP	Number of independent paths.

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

	NCN	TCT	NL	NP
level 1	0 or 1	0 or 1	*	*
level 2	0 or 1 2 to 5	GT. 1 *	* LE 2	* 201 - 400
level 3	2 to 5 GT 5 GT. 5	* * *	LE 2 EQ 0 1 - 3	1 to 200 LT 0.2*NCN**2 OR 0.2*NCN**2 - 0.8NCN**2 1-400
level 4	2-5 2-5 GT 5 GT 5	* * * *	LE 2 GT 2 1-3 GT 3	GT 400 1 - 500 401 - 1000 1 - 1000
level 5	2-5 GT 5 GT 5	* * *	GT 2 1 - 3 GT 3	GT 500 GT 1000 GT 1000

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

Table 2. Preliminary results obtained from the classification table.

The data illustrate the following points:

- The 4 selected static metrics are weakly correlated (less than .65) and can be taken as independent variables for the decision tree.

- The module's complexity 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 complexity definition. Metric values were not available to them.

- Module complexity 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 highly complex 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 level.

Figure 4 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.

One component of complexity that is not explored by this classification table is module data coupling. Research is currently under way on the data-flow aspects of the complexity of a module, and preliminary results are promising.

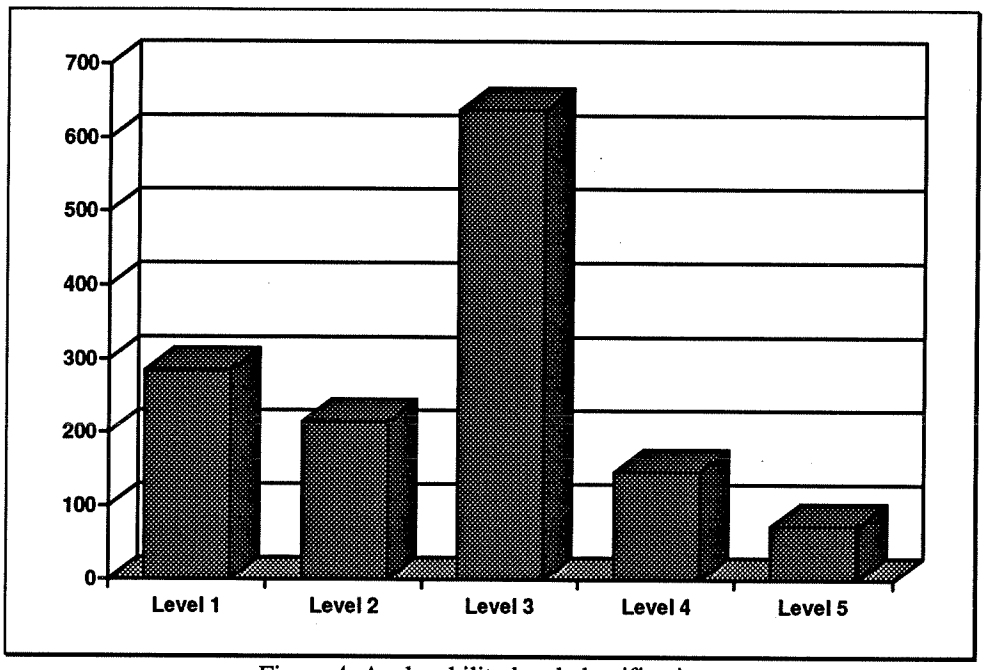

Figure 4. Analyzability level classifications

6. Control-flow representation.

Control-flow representation is a basic tool for visualizing and understanding program structure. This section presents a new control-flow representation that provides the following advantages: it is easy to visualize, it outlines the breaches of structure and it is programming-language-independent. (4)

There is some evidence that the visualization of control flow can provide a better understanding of the programming process. It can improve testing and validation by making a formal link between algorithm implementation and specifications. It can reduce maintenance by showing the complexity of the control flow. It can help to establish norms and standards in quality control programs and to define testing methods. It can provide a better understanding and definition of the cohesion and coupling of software units.(5)

Despite growing interest in software metrics, control flow representation is still empirical(6). An overview of some existing control flow representations illustrates the need for a more formal representation. Most current control graph representations do not precisely link graph features to specific source code statements. Some visual representations may be clumsy, and may become confused when programs get bigger, when, in fact, they are most needed. Figures 5 and 6 show typical examples of such control graphs.

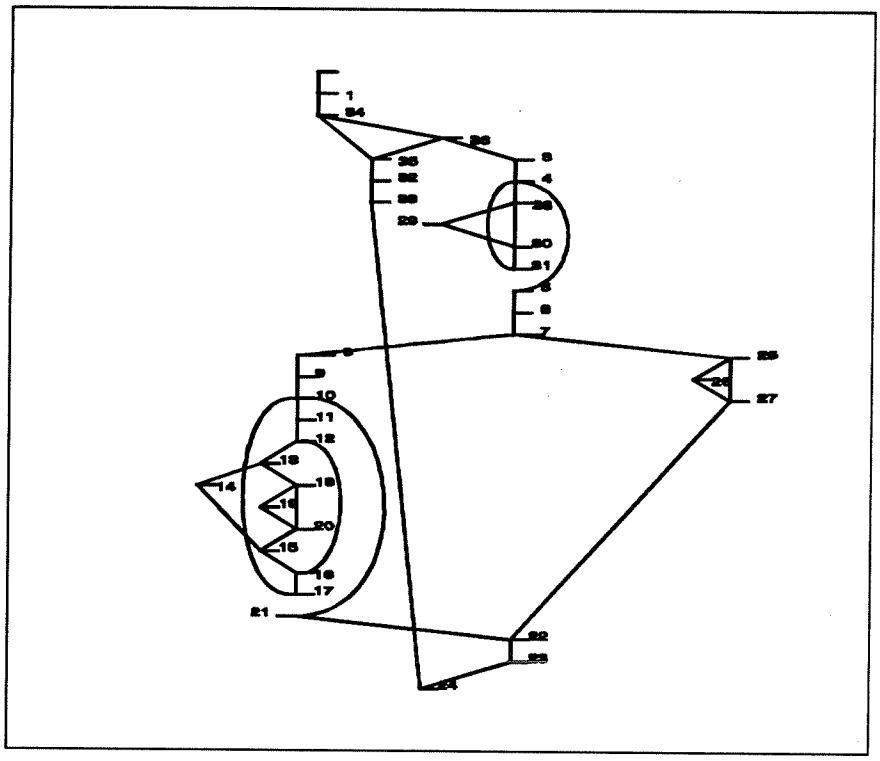

Figure 5: Example of control flow representation (7a)

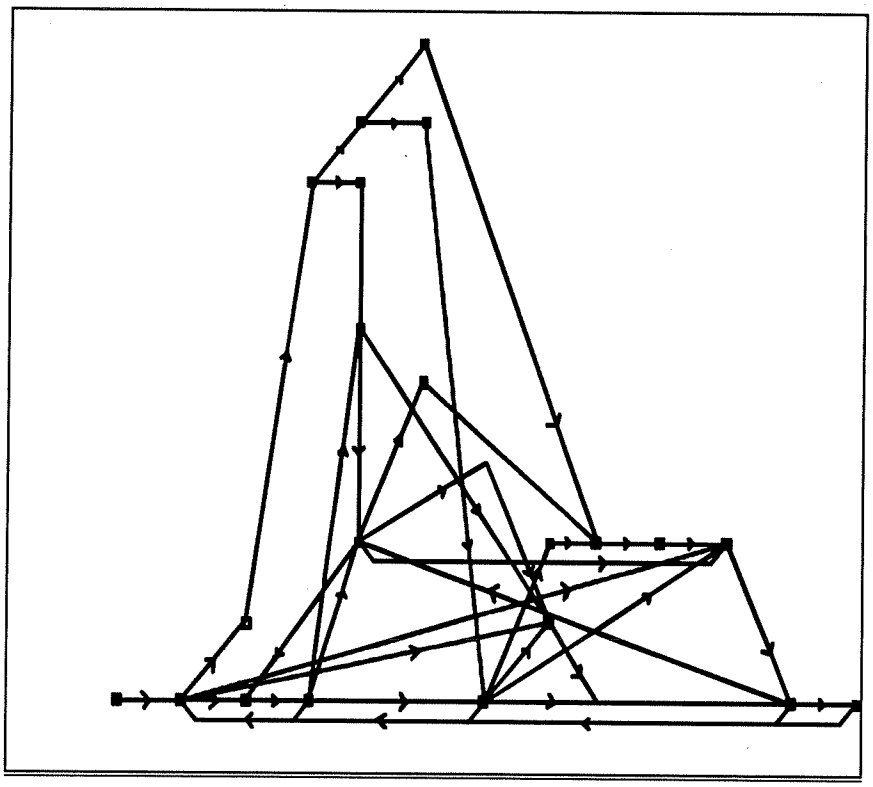

Figure 6: Example of control flow representation (7b).

Mathematical descriptions of program control flow are based on nodes and arcs. A node can be a predicate node, which represents a decision point, or a procedure node, which represents sequential statements. These approaches define construction rules working on an arbitrary set of primitives. Some authors are particularly concerned about the generalization of the structuredness notion. Others use such a mathematical model to study metrics' sensitivity using atomic modifications (8). This work leads to the problem of exact software control flow modeling: "As is well known, directed graphs have often been used to model control flow in sequential programs. The modeling process itself is a non-trivial task and has usually been underestimated (9).

Recent work on control flow modeling presents a formalization of the control flow representation (4). Programs are decomposed into basic blocks. Each basic block is represented by specific icons. Icons are assembled according to redefined rules. The process keeps all control-related information as it appears in the source code. The goal is to represent control flow as it is and not as it should be. The icons provide a visual representation that is independent of the size or complexity of the control flow. Some graphical features are:

discrimination between forward and backward flow,
identification of breaches of structure,
exact matching of the source code.

A study of automated diagram drawing gives some rules for good graphical representation (10). The authors state that "the lower the visual complexity is, the easier the diagram is to understand."

They suggest these two considerations: (1) Straight lines are easier to follow than curves. The origin and destination of a straight line are more direct than those of curves, and (2) consistency increases readability. Consistency means that all objects that have the same connectivity, semantics and geometrical structure are drawn in the same way. For example, each edge of one kind in a diagram should be of the same shape and thickness. Different kinds of edges in the same diagram should be of different shapes and thickness.

6.1 Basic block representation

Icons are used to represent a program's basic block. 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 (11). It is impossible to jump into a block or out of a block. 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, identifies connectors with 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 7A shows a basic block terminated by sequential connectors.

Figures 7B and 7C 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 7D 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 7E 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 that can be entered by a sequential or a label connector, or a block that can be exited by a sequential or a branching or

multibranching connector. Figure 7F shows a block that can be entered by a label or a sequential flow. Figure 7G shows a block that 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 7H shows the representation of a backward branching block with a sequential node and a double vertical line terminated by a branching.

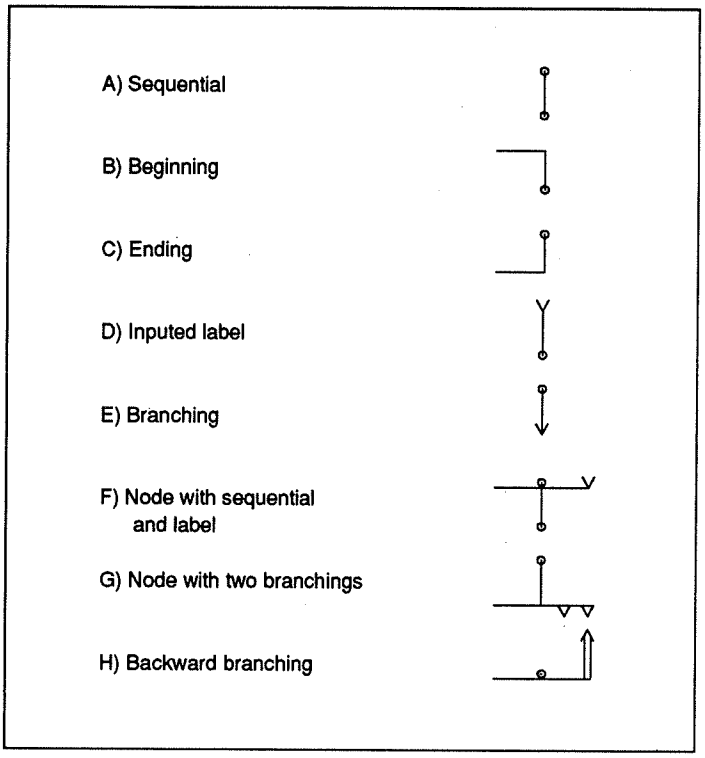

Figure 7: Basic block icons

The following association rules are used to group the resulting icons of a program together to represent the complete control flow.

1) Resulting icons are ordered according to the expression line-up. This is to maintain correspondence between the source code and the control flow).
2) Corresponding numbered branching or label connectors are joined together by extending or stretching connecting lines.
3) Vertical lines are moved horizontally to respect the following conditions:
   Lines must be to the right of the main axis to maintain half-plane representation,-
   Lines must not overlap,-
   The farthest target destination is the farthest from the main axis,
   Lines are kept as close as possible to the axis,
   Lines are moved horizontally or vertically by defined space increments.

Figure 8 shows all the basic blocks linked together according to the above rules.

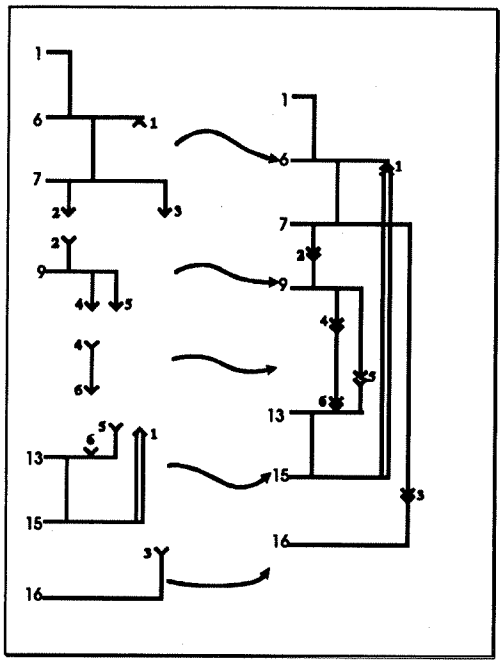

Figure 8: Association as applied in the example.

6.2 Applications

Figure 7 illustrates this new control-flow representation. The graph nodes are either branch instructions or branching addresses. The only exceptions are the program entry node, a dummy node inserted before the first nondeclarative statement (at the end of the declaration section, if there is one) and the main exit node.

Directed arcs link origin nodes to destination nodes. An arc may be of the sequential or branch type. 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 weight of an arc is the number of executable statements associated with the arc. The node number is a pointer to the line statement in the source code.

Information can be added to emphasize the relationship of the graph to the source code. Horizontal arrows differentiate entry and exit nodes from pending nodes. Arcs are named according to their control flow type. Line numbers corresponding to source code statements can be associated with each node. Figure 9 shows the final representation. A weight can be associated with each arc in order to represent the number of sequential statements (basic block S) included in it.

Half-plane representation allows the identification of arc crossings (12). The crossings that violate the rules of structured programming are called breaches of structure, and are shown by square dots in the final representation.

This graphical representation of the control flow corresponds to the general interpretation of graph theory (13). Arcs contain statements from the source code. Nodes are the logical states of a program where arcs go from one node to another.

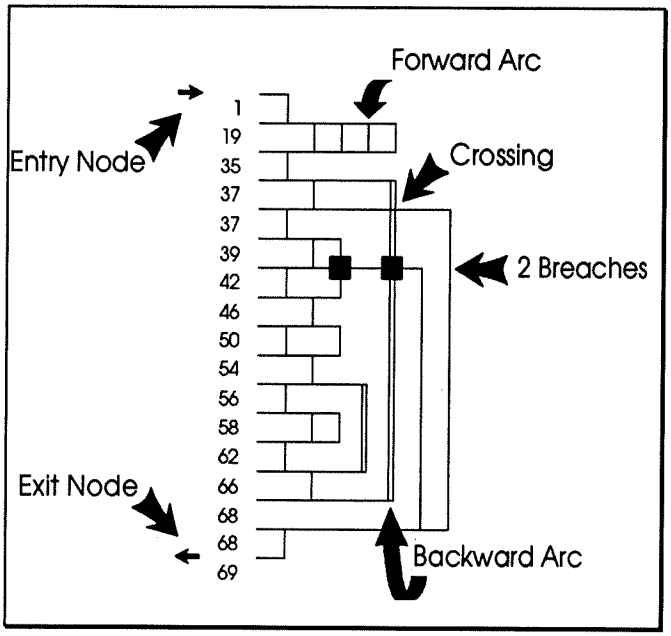

Figure 9: Final representation of the control graph.

Figure 10 presents the iconic control flow graph equivalent to the representation shown in Figure 6. It is easy to see that the program is well structured and that there is an exit statement in the code (line 11). Figure 10 provides added information by indicating a weak control flow somewhere between lines 22 and 27. These two parts of the program are linked by a single basic block. Detailed data flow analysis may indicate two independent functionalities. This may be indicative of low functional cohesion. These characteristics are not obvious from Figure 6. The iconic control flow graph represents the real control flow of the source code as it is written, and is not an interpretation of it.

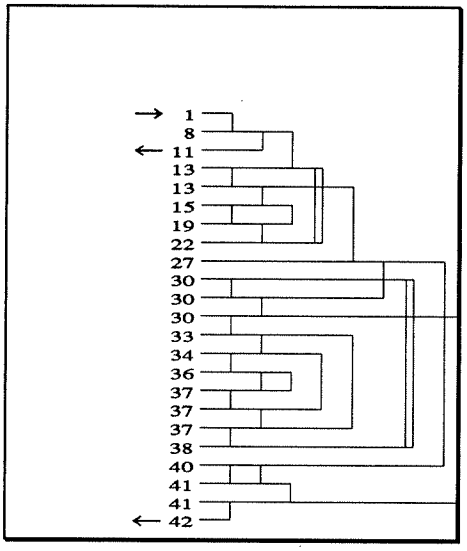

Figure 10: Control graph equivalent of Figure 6.

Control graph construction is straightforward and easy to apply to small programs. Larger programs need a software tool to compute and draw the control graph. DATRIX^{TM} is used to evaluate millions of lines of code of large projects undergoing code inspection, testing, production approval and maintenance. The

objective of the tool is to provide understandable measurement and lucid visual information. This tool enables the user to toggle back and forth between the control graph and the source code. It also computes a set of metrics related to a software unit's syntax, control flow and organization.(14). Figure 11 shows an example of a control flow graph computed by DATRIX^{TM}. The line organization on the graph is kept constant and is independent of the size or "complexity" of the control flow.

Such work was needed for various reasons. Control flow visualization is useful in helping programmers develop and maintain their programs. A great deal of work is being done on the transformation of control flow (8,9) in re-engineering, restructuring, documentation, optimization, etc. The solution presented is a simple one, and can be applied to any procedural programming language. A software tool has been implemented and the power of the approach demonstrated. The most useful metrics computed on the representation are the number of independent paths, the number of decision summits, the nesting levels and the number of breaches of structure.

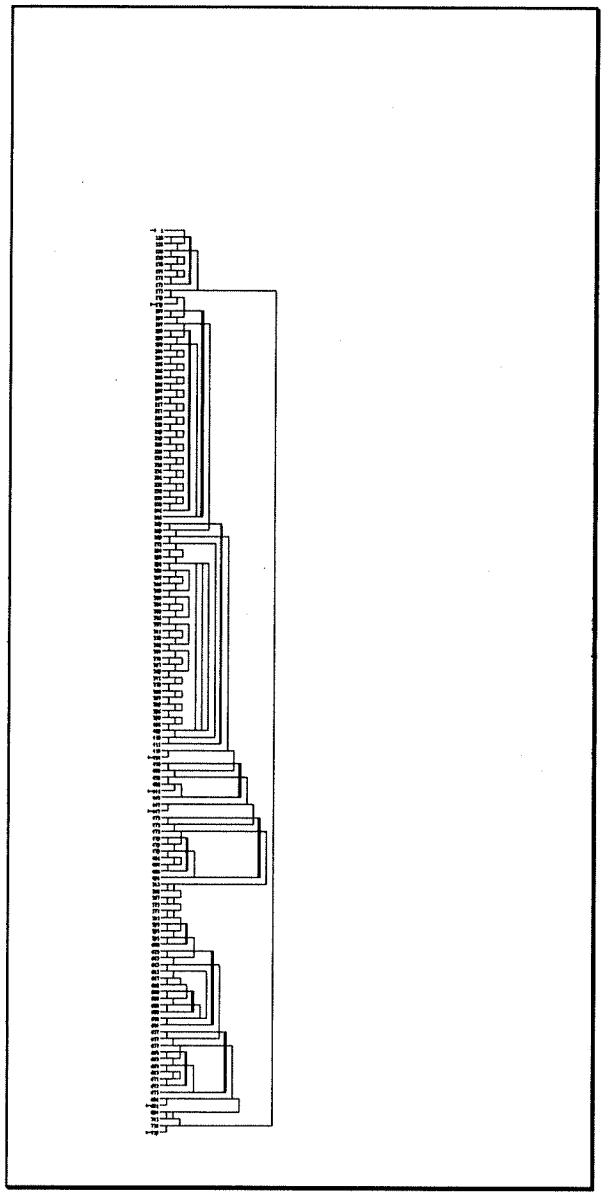

Figure 11: Extended DATRIX^{TM} control flow representation

7.Restructuring modules

This section shows how the control flow constructed from static analysis of the source code could be used to restructure a module. Considering previous evaluations, modules that need to be restructured are identified. The restructuring transformation involves either the introduction of predicate flags or code duplication or both. The detection of unstructuredness is based on the six basic forms of unstructuredness proposed by Oulsnam (15)

The following figures (11,12 and 13) present the source code of an unstructured flowgraph, followed by its original and structured flowgraphs. The example presented is a small search function in a sorted array.

```
int lookup( int item, int table[], int *finish ) {
int start = 1;
int i, ret;
label_a:
i = (start+ *finish)/2;
if(item == table[i]) goto label_b;
if(table[i]<item)  start = i+1;
if(table[i]>item)  *finish = i-1;
if((*finish-start)>1) goto label_a;
if(table[start]==item) goto label_b;
if(table[*finish]==item) goto label_b;
ret=0;
goto label_c;

label_b:
ret=1;

label_c:
return ret;}
```

Figure 11. Source code example

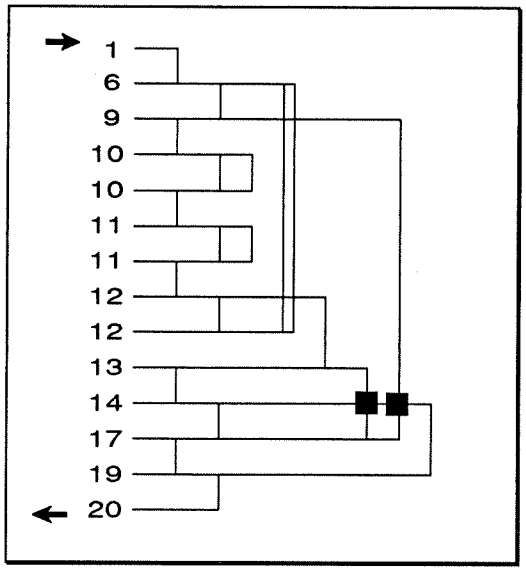

Figure 12. Unstructured flowgraph.

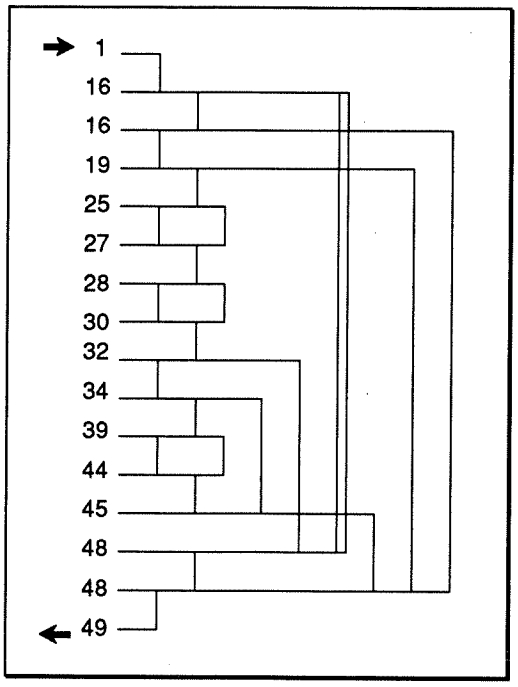

Figure 13. Restructured flowgraph.

8. Concluding remarks

This paper presents examples of various applications of static metrics. Areas where more research is needed are outlined.

Metrics are computed from the static analysis of source code. Statistical analysis is performed to identify the information content of the metrics selected. All, or a subset, of the metrics are selected to perform the assessment. Metrics' distributions are studied to determine the usual ranges of values, and out-of-range functions are identified. Percentile profiles give a project overview of the percentage of out-of-range functions. Quality factors are defined by grouping metrics. A normality profile gives the percentage of success for every quality factor. Finally, function coupling is evaluated from the 3-D call graph representation.

The iconic control flow representation helps to precisely visualize the control flow of a program. This visualization is rigorous and always presents the same information density. The iconic control flow representation could be a reliable and efficient way to manage control flow complexity. The most useful metrics computed on the representation are the number of independent paths, the number of decision summits, the nesting level and the number of breaches of structure.

Such work was needed for various reasons. Control flow visualization is useful in helping programmers develop and maintain their programs. A great deal of work is being done on the transformation of control flow in re-engineering, restructuring, documentation, optimization, etc.. Also metric measurement needed a more rigorous framework. New implementation paradigms needed to be evaluated. Hopefully this goal will be reached by third generation static analyzer tools.

Research is currently under way to identify new metrics and to increase the information content of the control graph. The formal basis of control flow representation enables research on control flow optimization, automated reverse engineering and the automated evaluation of control flow based on algorithm specifications.

This work deals strictly with the internal control flow of a software unit, which constitutes only a part of the information contained in the source code. Other information, such as software unit calls and data flow, can be studied in a similar way.

This approach, based on graphs and profile, is visual and as a result many different types of people are afforded immediate access to information. Function profiles can be modified at will to explore specific aspects of program quality, and profile simulations can evaluate the impact of function redesign on project quality. The programmer can obtain immediate feedback on any implemented function.

The versality of this approach makes it conducive to exploratory study and to the development of custom assessment programs. The metrics based on information theory can be integrated into exiting metrics. This approach is also of interest in the maintenance and testing area where an inside picture of the software can provide insightful clues as to where further the development effort should be concentrated. We feel that merely introducing this tool into the development process at an earlier stage may result in increased productivity

The problem of metric validation remains. This could be solved by an extensive study of the relationship between the metrics presented and data on error rates, development costs and reliability indicators. Such a study 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.

Acknowledgements

We thank members of our group who have worked on these projects, and offer special thanks to André Beaucage, Jean Mayrand, Frederick Chouinard and Martin Leclerc. Support for this work was provided in part by Bell Canada, the National Research Council of Canada (under grant A0141), Schemacode International inc., and the various organizations that apply these approaches and provide us with meaningful feedback.

References

1. D. Coupal and P. N. Robillard, "Factor Analysis of Source Code Metrics," *Journal of Systems and Software*, Vol. 12, No. 3, pp. 263-269, July 1990.
2. Robillard, P. N., "Schematic Pseudocode for program constructs and its computer automation by SCHEMACODE", *Communications of the ACM*, Nov. 1986, Vol 29, no 11, pp 1072-1089.
3. J. C. Munson and T. Khoshgoftaar, "The Dimensionality of Program Complexity," *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, pp. 245-253, May 1989.
4. Robillard, P.N., Simoneau, M., "A New Control Flow Representation", *15th Annual Int'l Computer Software and Application Conf. (COMPSAC91)*, Tokyo, Japan, Sept. 1991.

5 Robillard, P.N., Coallier, F., "Practical Experiences With Source-Based Measurement", *International Conference on Applications of Software Measurement*, pp. 43-47, Nov. 12-15, 1990, San Diego.

6 Tripp, L. L. "Bibliography on Graphical Program Notations," *ACM SIGSOFT*, Vol. 14, No. 6, pp. 56-57, 1989.

7a Verilog, Logiscope General Presentation, Verilog U.S.A., Alexandria, Virginia, October 1988, p2,4.

7b McCabe & Associates, The Analysis of Complexity Tool, McCabe & Associates Inc., Columbia, Maryland.

8 Zuse, H. *Software Complexity - Measures and Methods*, Walter de Gruyter & Co., Berlin, 1991.

9 Fenton, N. E. and Mole, P. D. A. "A Note on the use of Z to specify flowgraph decomposition", *Journal of Information and SoftwareTtechnology*, 30(7), pp. 432-437, 1988..

10 Ding, C., Mateti, P. "A Framework for the Automated Drawing of Data Structure Diagrams", *IEEE Transactions on Software Engineering*, SE-16(5), pp. 543-557, 1990.

11 Aho, A. V., Sethi, R., Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1988

12 Woodward, M. R., Hennell M. A. and Hedley, D. "A Measure of Control Flow Complexity in Program Text", *IEEE Transactions on Software Engineering*, SE-5(1), pp. 45-50, 1979.

13 Berge, C. *Graphs*, North-Holland Mathematical Library, North-Holland, 1985.

14 Robillard, P. , Coupal, D. and Coallier, F. "Profiling Software through the Use of Metrics", *Software - Practice and Experience*, 21(5), pp. 507-518, May 1991.

15. Oulsnam, G. "Unravelling Unstructured Programs", *The Computer Journal*, Vol. 25, No. 3, 1982.