Static and Dynamic Software Quality Metric Tools

By Kevin A. Mayo, Steven A. Wake and
Sallie M. Henry

TR 90-52

Static and Dynamic Software Quality Metric Tools

Kevin A. Mayo Steven A. Wake
Sallie M. Henry

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

(703) 231-6931

mayoka@vtopus.cs.vt.edu
wakes@vtopus.cs.vt.edu
henry@vtopus.cs.vt.edu

Fax: (703) 231-8456

Abstract

The ability to detect and predict poor software quality is of major importance to software engineers, managers, and quality assurance organizations. Poor software quality leads to increased development costs and expensive maintenance. With so much attention on exacerbated budgetary constraints, a viable alternative is necessary. Software quality metrics are designed for this purpose. Metrics measure certain aspects of code or PDL representations, and can be collected and used throughout the life cycle.[RAMC85]

Automated software quality measures are necessary for easy integration into the software development process. This research reports on two metrics: Interface Metrics and Dynamic Metrics. Interface metrics provide the capability to measure the complexity of communicating modules. Dynamic metrics provide a measurement of the software system quality as it is executed during the validation and verification life cycle phases.

Keywords: Software Quality, Software Quality Metrics, Software Quality Tools.

Introduction

As fast as computer science solves one problem, another comes up. Yet, through this entire process the 'art' of managing software development is still vague. One prime aspect of the development process that must be controlled is the cost. Many horror stories exist about development fiascos, but there are some methods and tools that can help alleviate these problems.

One such method involves the use of software quality metrics throughout the lifecycle. Software quality metrics are measurements that are derived from designs or source code. These measurements help decide the flow of projects, and thus can have a great impact on the cost. These measurements can aid in the decision among designs, locate overly complex designs or code that will lead to costly debugging or maintenance, and reflect an overall reliability.

Calculating these metrics by hand is a long and tedious process which is not cost effective. Therfore we must automate the collection of software quality metrics. A tool exists which collects the necessary information to calculate these metrics for a software project .

Background

There has been a great deal of interest about SQA metrics in the literature, with many different proposed measurements. These metrics gather information in order to quantify certain aspects, and because these are generated over non-executing code they are classified as static. To date the proposed static SQA metrics have fallen into one of three categories: code metrics, structure metrics, and hybrid metrics.

Code metrics are generated from token counts from a parse of the source language. This source language can be computer programming language: e.g., FORTRAN, Pascal, C, Ada. Or, this source language can be a programming design language, either textual or graphical. The most common metric of this category is source lines of code (SLOC, or 1000 SLOC = KSLOC). This measurement has been around for as long programs have been written. Other measurements include: Halstead's Software Science, and McCabe's Cyclomatic Complexity. [HALM77] [MCCT76] These are by no means the totality of static SQA metrics, however, these are used in this study.

Structure metrics measure the interconnections of the source code elements. Simply, these are measurements of control flow through communicating source code elements. One such measurement is information flow as defined by [HENS81]. This measurement, INFO, is a function of the number of calls into and out of an element, giving an indication as to the elements usage.

Hybrid metrics are the best of both code and structure SQA metrics. Using aspects of both code and structure metrics these measurements capture more information about the element under scrutiny. These, however, are very costly to generate for a given source code element or design.

The reader is directed to the Appendix for an introduction to the static SQA metrics discussed within this section.

A program is a dynamic entity. During execution it has the effects of calls, return from calls, variables set, loops executed and dynamic data structures developed. Static metrics can not be expected to take into account the effects of this dynamic execution. In order to measure this it is necessary to have a set of dynamic metrics. These metrics provide an indication of the coverage of the program along with being useful for reliability modeling.

Interface Metrics

While the selection of established metrics does offer several meaningful measurements, the quantification of element communication has eluded researchers. Static SQA metrics were designed by this research to meet this need. Interface complexity metrics, ICM as they are known as, attempt to capture a meaningful quantification of the communication interface between two elements.

Element, or module, communication has been addressed within the literature. Stevens defined coupling and cohesion levels to help classify the interfaces between communicating elements [STEW74]. In order to achieve the loosely coupled and highly cohesive system structure that is ideal, the use of low complexity interfaces between elements is necessary. The interface complexity metrics allow a measurement of each interface between all communicating elements.

The previous metrics have fallen short of this goal. However, aspects of the established metrics are considered within the interface complexity metric. For instance, the INFO metric captures complexity associated with dataflow, however, this metric ignores the inherent complexities associated with differences in the data structures or size. McClure, in [MCCC78], examines the communicational environment classifying according to nesting within either selection or repetition control clauses, but ignores the varying degrees within each of these clauses.

The interface complexity metric measures both the environment and the data used in the communication.

Interface Metric Definition

As stated before, the interface complexity metric measures both the environment and the data associated with each communication between elements. In order to measure these aspects, many underlying measurements were defined and used in the definition of the interface complexity metric.

Data Complexity

First lets consider the data being used in the communication. What features of data makes one data object more or less complex than another? The data type is the most apparent differentiation of data objects that exists. Clearly, passing an integer is far less complex to comprehend than passing a linked-list of records. Hence, a quantification scheme for data types was created. A *type complexity* is therefore associated with each variable within the communication.

The data that is passed between two communicating elements, however, might not be a simple data object. The other alternative is to pass an expression, e.g., X + 3, Y * 2, etc. Therefore this expression must be analyzed and quantified as to its complexity, i.e. an *expression complexity* must be calculated. This expression complexity is a function of both the type complexity and the complexity associated with each operator, which is known as the *operator complexity*.

With the type and expression complexity the data complexity within the interface can be measured. As a side bar this research also used the extent of modification on each variable within the interface, however, this showed to be statisticly non-different.

Environment Complexity

The environment complexity is calculated by associating weights to the different selection or repetition structures found in the source code. This association of weights generates the *flow-control complexity*. Each of the different selection (IF-THEN, IF-THEN-ELSE, CASE, etc) or repetition structures (Test-Before, Test-After, Infinite, etc.) is given a weight. This gives a measurement at each line within the source code, which is used to calculate the environment complexity at the point of communication.

Relationship Among Complexities

Figure 1 demonstrates the way the measures combine into other measures. As mentioned before, the interface complexity measure contains both the data and environment measures. The data complexity is built up from the type and operator complexities. If the data is a simple variable, and not an expression, then the operator complexity drops out of the function. The environment complexity is built up from the expression and flow-control complexities.

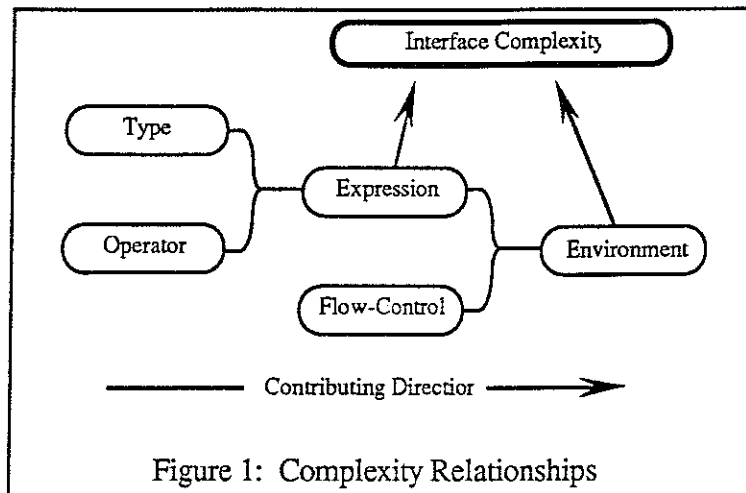

Figure 1: Complexity Relationships

All these complexities are contingent upon a pre-defined set of weights that the manager or software engineer creates. This gives the ability to change/modify the weights to meet their specific needs. Obviously users examining a real-time system are concerned with different aspects verses users examining a life critical system (i.e., speed verses software safety).

Interface Metric Tool

The process of generating the interface complexity metric is automated. In fact, if this process was not automated than the calculation of interface complexities even for short and

simple code would be difficult at best. The tool that calculates these numbers, as well as others, is depicted in figure 2. This tool works over several languages, i.e., FORTRAN, Pascal, C, and Ada. However, the calculation of the interface complexity measures are only implemented for the Ada language.

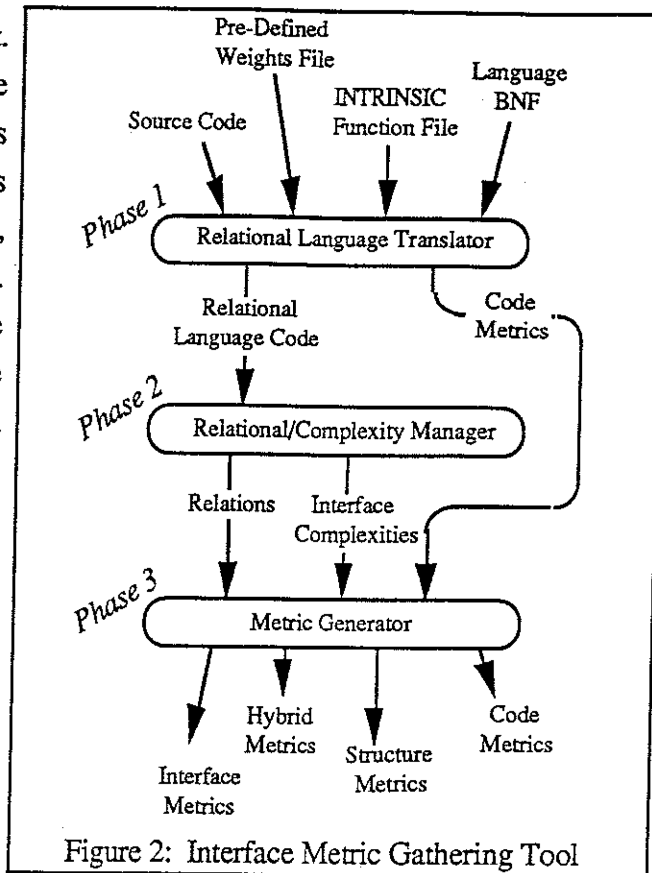

Figure 2: Interface Metric Gathering Tool

This tool utilizes a representation known as the relational language [HENS88]. This language contains all the primary constructs found in procedural languages. However, this representation does not maintain enough information to reverse engineer the algorithms or data structures. Therefore, a translation from a program source to relation language will generate a representation that can be useful for metric calculation, but useless for reverse engineering.

The metric analyzer, as this tool is known as, is a three phase process. The first phase is the only language dependant pass of the system. Therefore, it is possible for the first phase to be executed away from Virginia Tech, thus insuring the security of proprietary code. In other words, the analysis of the code beyond phase 1 of the tool does not require the source code. The first phase takes in the source code and pre-defined type and operator weights and generates the code metrics and a relational language representation of the source. Code metrics are generated in this phase because they are source dependant.

The second phase converts the relational language representation into a set of relations that maintain flow information. These relation will be used in the third phase. The second phase also calculates all the interface complexity information. This information includes metrics for all communicating elements (both environmental and parameter measurements), a breakdown of variable usage, and a user document. While the third phase is used for

viewing the data, the user document is generated to view the "nuts-and-bolts" of the system and the measurements. This document shows all variables, their type complexities, their usage (local, global, parameter), and all the separate interface costs between elements. This is very useful when fine tuning the weights to a specific need.

The third phase of this tool generates structure and hybrid metrics. This phase is also designed to display the information in a meaningful manner. With a relatively complex system the number of modules/elements could prohibit the cost effectiveness of examining each individually. This phase facilitates the grouping of elements into a module level.

Interface Metrics Results

The metric analyzer was used to analyze an 85 KSLOC Ada program. This program was developed by Software Productivity Solutions (SPS) of Melbourne, Florida. This system was sufficiently large and complex enough to test the interface complexity measurements.

The results are very promising, and they are discussed in [CHAB90]. In summary, these metrics can be used from the design phase on, and can indicate possible areas of poor code quality.

Dynamic Metrics

Static metrics are derived from an analysis of non-executing code. Dynamic metrics are derived from an analysis of the code while it is executing. They provide an indication of what calls are actually taking place, the number of statements executed, and what paths are being executed.

Dynamic Metric History

There are several different methods for performing analysis of an executing program. Most of these methods are used to do performance analysis to determine what parts of a program can be speeded up to decrease overall execution speed. These same methods work for gathering dynamic information about a program. These techniques include profilers [GRAS83] [BISM87], execution monitors [PLAB81], run time langauages [COHJ77], instruction counts [MCDG82], traces of execution [SHRA84], and program slicing

[WEIM84]. These techniques can be implemented by preprocessing the source code, postprocessing the assembly code, or having separately executing monitor programs.

Dynamic Metric Definition

Dynamic metrics include both complexity measures and measures useful in reliability modeling. Dynamic metric values are dependent on the input or test data with which the software system is run. This could provide different values when different test suites are run against the software system but will yield a consistent measure if the same test suite is run multiple times.

Dynamic metrics and the reliability model associated with them are useful in certifying a product as ready for release in terms of reliability. They could also provide information about the complexity and the reliability of a prototypes of a product to help determine further directions which should be taken concerning the prototype.

The first dynamic metric is Frequency which is the number of executions of a module. This measure is used for reliability purposes. Module execution without error indicates less likelihood of an error in this module while an error indicates less reliability. Duration, the time required to execute a section of code can be used similarly in a reliability model.

Parameter reference count and variable reference count are the number of times the parameters to a module and the variables within a module are referenced. In a static sense, the more times these are referenced in module the more is known about the parameter or variable similar to Woodfield's Review Complexity [WOOS82]. However, dynamically there may be many references where there are few static references. Many references would increase complexity due to the increased interactions between the variables and parameters.

Loop Executions is similar to frequency but counts the number of iterations of a loop.

Static structure metrics weight all possible paths equally. By doing a dynamic analysis of branch selections a Branch Selection Percentage can be calculated and used for weighting of static structure metric values which count all branches equal.

Call Nesting Depth measures the depth of calls made to get to current module. This measures how much must be kept track of to know what is going on in the current module. Knowledge of the environment when the current module is called is necessary when considering the effects of this module. The deeper the nesting level, the more things must be taken into account and thus a more complex environment must be considered.

Called function count is similar but counts the number of modules which are called from this one. It is an indication of how many other modules are affecting this one. Higher complexity is expected when there are more modules called.

Statements executed is the dynamic equivalent to lines of code. It is a basic measure of complexity based on how big the software system is in dynamic terms.

Dynamic Metric Tool

A static metric analyzer already exists at Virginia Tech which determines metric values from analysis of the source code. This analyzer can be adapted to handle dynamic metrics.

A preprocessor would modify source code by adding some profiling commands. These commands would be inserted code which would perform counts and other forms of dynamic analysis. The new code would be executed when the program is run and the results could then be analyzed with the use of the existing metric analyzer or a slightly adapted version of the analyzer.

Conclusions

This paper discusses two different realms of metrics: static and dynamic. While each one is interesting, it is best to view both classes of metrics when analyzing a system. Static metrics characterize system attributes from design through maintanence, and dynamic metrics from coding through maintanence. The SQA interface complexity metric has been shown to view the communicational cost of software. Various dynamic metrics are proposed here, and they look hopeful at best.

It is important to note that these metrics should be viewed on a whole by a software engineer or manager. We believe that these metrics should be used as part of a complete management scheme or methodology tailored to the particular development environment.

This scheme would cover the entire software development life cycle, not concentrating on the coding phase alone. This provides for collection of the metrics from the beginning of the life cycle. In other words, a design is automatically analyzed, and provides a basis for choosing among several alternate designs for a single product. This helps reduce costs, and hopefully decrease maintenance activity.

Designs, in proper and readable formats, can be analyzed, thus these measurements can span the lifecycle from design to maintenance.

Bibliography

[ADRW82] Adrion, W.R., Branstad, M.A., Cherniavsky, J.C.,"Validation, Verification, and Testing of Computer Software", *ACM Computing Surveys* June 1982, pp. 159-192.

[BASV85] Basili, V.R., Selby, R.W., "Calculation and Use of An Environment's Characteristic Metric Set", *8th International Conference on Software Engineering*, 1985, pp 386-391.

[BISM87] Bishop, Matt, "Profiling Under UNIX by Patching", *Software Practice and Experience:*, October 1987, pp. 729-739.

[CHAB90] Chappell, B., Henry, S., and Mayo, K., "Measurement of Ada Throughout the Software Development Life Cycle", *Proceedings of the Eighth Annual Conference on Ada Technology*, 1990, pp. 525-532.

[COHJ77] Cohen, J., Carpenter, N, "A Language for Inquiring about the Run-time Behaviour of Programs", *Software Practice and Experience*, July 1977, pp. 445-460.

[ELSJ84] Elshoff, J.L., "Characteristic Program Complexity Metrics", *7th International Conference on Software Engineering*, 1984, pp 288-293.

[GRAS83] Graham, S.L., Kessler, P.B., McKusick, M.K., "An Execution Profiler for Modular Programs", *Software Practice and Experience*, 1983, pp. 671-685.

[HALM77] Halstead, M.H., *Elements of Software Science*, New York, Elsevier North-Holland Publishing Co., 1977.

[HENS81] Henry, S.M., Kafura, D.G., "Software Structure Metrics Based on Information Flow". *IEEE Transactions on Software Engineering*, September 1981, pp. 510-518.

[HENS88] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or, Do You Recognize This Well-Known Algorithm?", *The Journal of Systems and Software*, Volume 8 Number 1, January 1988, pp 3-11.

[KAFD85] Kafura, D., Canning, J., "A Validation of Software Metrics Using Many Metrics and Two Resources", *8th International Conference on Software Engineering*, 1985, pp 378-385.

[MCCT76] McCabe, T.J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, December 1976, pp. 308-320.

[MCDG82] McDaniel, G., "An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies", *SIGPLAN Notices*, April 1982, pp. 167-176.

[PLAB81] Plattner, B., Nievergelt, J., "Monitoring Program Execution: A Survey", *IEEE Computer*, November 1981, pp. 76-93.

[RAMC85] Ramamoorthy, C.V., Tsai, W., Yamamura, T., Bhide, A., "Metrics Guided Technology", *Proc. Computer Software and Applications Conference*, 1985, pp 111-120.

[SHRA84] Shroeder, A., "Integrated Program Measurement and Documentation Tools", *7th International Conference on Software Engineering*, 1984, pp. 304-313.

[WAKS88] Wake, S.A., Henry, S., "A Model Based on Software Quality Factors Which Predicts Maintainability", *Proc. Conference on Software Maintenance - 1988*, Oct. 1988, pp 382-387.

[WEIM84] Weiser, M., "Program Slicing", *IEEE Transactions on Software Engineering*, July 1984, pp. 352-357.

[WOOS82] Woodfield, S.N., Shen, V.Y., Dunsmore, H.E., "A Study of Several Metrics for Programming Effort", *The Journal of Systems and Software*, Volume 2, Number 2, June 1982, pp. 97-103.

Appendix — Static SQA Metrics

This Appendix contains brief descriptions of the metrics that are discussed within this paper. These metrics fall into one of three categories: code, structure, or hybrid.

This appendix defines the metrics that are used within this research:

LOC:	Lines of Code
N, V, E:	Halstead's Software Science Indicators
CC:	McCabe's Cyclomatic Complexity
WOOD:	Woodfield's Review Complexity
INFO:	Henry-Kafura's Information Flow Metric

Lines of Code (LOC)

This metric, most probably the oldest, is an enumeration of the lines of code. Simple, but indicative of the complexity. There is, however, some debate what constitutes a line of code from one language to the next.

Halstead's Software Science (N, V, E)

Halstead introduced a series of measures based on the countable aspects of source code. Software Science [HALM77], measures are based on:

n_1 = The number of unique operators
n_2 = The number of unique operands
N_1 = The total number of operators
N_2 = The total number of operands

$n = n_1 + n_2$ = Size of Vocabulary
$N = N_1 + N_2$ = Length of the Program

From the manipulation of these four countable measures, Halstead created software science indicators (among others) for: Volume (V), Program Level (L), Difficulty(D), and Effort (E).

The volume of a program is defined, by Halstead, to be a function of the length (N) and the vocabulary size. The volume represents the storage requirements of the program, and is defined as:

$$V = N * Log_2(n)$$

The $log_2(n)$ component of the equation represents the storage requirements of the vocabulary symbols.

The volume measure can change depending upon the size of the algorithm chosen for the task. If there are *n* different algorithms (and thus *n* different volumes) it can be shown that there is a minimum volume over these code sections. This minimum volume, or potential volume of the most concise algorithm (V^*), is defined but can never be calculated due to its theoretical nature. With this, Halstead defines the program level to be:

$$L = \frac{V^*}{V}$$

Because V^* can never be calculated an estimator must be attained:

$$L' = \frac{(2 * n_2)}{(n_1 * N_1)}$$

The difficulty of an algorithm is defined in terms of elementary mental discriminations (EMD). EMD represents the number of comparisons required to implement the algorithm with respect to the given language. Therefore, Halstead defined it as the reciprocal of the program level estimator:

$$D = \frac{(n_1 * N_1)}{(2 * n_2)} = L^{-1}$$

Halstead's effort measure is a function of the difficulty and the volume. It represents the mental effort required to design the algorithm from conception to implementation. Effort (E) is:

$$E = V * D = V * L^{-1} = \frac{V^2}{V^*}$$

Effort, Difficulty, Program Level, and Volume are the measures that compose the Halstead Software Science indicators.

McCabe's Cyclomatic Complexity (CC)

McCabe's cyclomatic complexity metric is built upon the selection and repetition structure of the code. [MCCT76] In the model, code is represented by a strongly connected graph G. The nodes represent the code and the arcs represent the control flow. If the code is a selection statement, then several arcs may extend from that node. This graph maintains the path or flow information of the algorithm and has a unique entry and exit point. The graph is strongly connected since there is an arc from the exit to the entry. With this framework, McCabe defined:

$$V(G) = E - N + 2 = \text{Cyclomatic Complexity, where:}$$

E = Number of Edges in graph G
N = Number of Nodes in graph G

V(G) represents the number of control paths in the algorithm. McCabe suggested an upper limit of V(G)=10, stating that this bound would curtail program complexity [MCCT76]. It should be noted that in this calculation, V(G), is the cyclomatic number found within classical graph theory, and it is the maximum number of linearly independent circuits or paths within G.
The calculation of this metric may seem, at first, to be complex. Yet, the generation of this metric reverts to the enumeration of the decisions within the code plus one [WOOM79]. These decisions include the selection and repetition constructs. Also, the total complexity of the set of modules (the program) is equal to the sum of all module V(G)'s.

Henry-Kafura's (INFO)

Information flow is the basis for the structure metric defined by Henry and Kafura [HENS81]. Here the complexity is defined in terms of the relationship of a procedure to its environment. This relationship represents the flow of information into (fan-in) and out-of (fan-out) a module. The terms are defined as:

Fan-in — The number of local flows into the procedure plus the number of data structures from which the procedure retrieves information.

Fan-Out — The number of local flows from the procedure plus the number of data structures that the procedure updates.

With the above definitions defining the flow relationship, the complexity of procedure p was defined to be:

$$C_p = (\text{Fan-in}_p * \text{Fan-out}_p)^2,$$

Fan-in_p = Fan-in for procedure p
Fan-out_p = Fan-out for procedure p

Where ($\text{Fan-in}_p * \text{Fan-out}_p$) represents the number of all possible data paths into and out of procedure p, and it is squared because the complexity relationship is not linear. This complexity points out areas (procedures) with very high information correspondence, and thus a location for errors and error propagation.

The information flow metric was introduced within the scope of the structure metrics; however, this metric scheme also can be modified to be of a hybrid type. In this case the metric is defined as:

$$C_p = C_{ip} * (\text{Fan-in}_p * \text{Fan-out}_p)^2, \text{ where:}$$

C_{ip} = Internal complexity of p
Fan-in_p and Fan-out_p are defined as before.

The hybrid form of the Henry-Kafura metric is used within this study as INFO.

Woodfield's Review (WOOD)

Woodfield's review complexity defines the complexity associated with programmer time. The assumption is that certain modules must be addressed several times due to their interconnections within the program. The interconnection scheme is defined with respect to module connections of which there are three different types:

$A \Rightarrow_c B$	Control Connection: Module A invokes module B
$A \Rightarrow_d B$	Data Connection: Module A modifies some variable that is used within module B
$A \Rightarrow_i B$	Implicit Connection: Assumptions made within module A are used within module B.

If any of these three types of connections hold, then there is a connection from module A to module B. These connections form:

Fan-in_i The number of times that a module needs to be reviewed. This is made up from a combination of the connections for module i.

It should be noted that the idea of Fan-in here differs from the Henry and Kafura model by including the aspect of implicit connections.

With these definitions in mind, Woodfield continued on to build a rating called the Review Factor:

$$RF_i = \sum_{k=1}^{\text{Fan-in}_i} RC^{k-1} \qquad RC = \frac{2}{3}$$

Where RC is the review constant that has been used by Halstead [HALM77].