Dynamic Software Metrices

Erich Schikuta

CRPC-TR93361
November 1993

Center for Research on Parallel Computing
Rice University
P.O. Box 1892
Houston, TX 77251-1892

DYNAMIC SOFTWARE METRICES

Erich Schikuta[1]

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston , TX 77251-1892

Abstract

In this paper a dynamic approach to measure the coupling of software systems is proposed. The conventionally used static measures are only limited suitable for the evaluation and characterization of such systems. We extend the static methodology with a dynamic component and define new measures based on this approach. A model for system characterization based on varying execution profiles is given. The layout of an automated monitor system is presented, which can be directly integrated into a software development system.

1. Introduction

Measures based on module coupling to assess the quality of software products are widely used in the software design and development process. The term coupling defines the strength of associations established by connections from one software module to another ([Stevens74]). In the literature many different static approaches to measure the coupling degree of software systems were proposed, as in [Myer76], [Yourdon79], [Card90], [Henry81], [Fenton90], etc.

In this paper we present a dynamic framework to measure the coupling of software systems. We introduce dynamic characteristics, program execution profiles and a program monitor system.

The measurement of characteristics of a software system is similar to the task to assess or measure the rate or type of intelligence of a human being in psychology. Of course, the number and complexity of the brain gyruses is correlated with the intelligence, but nobody would calculate the intelligence quotient out of these factors. The IQ is calculated by the behavior and reaction of the brain. This measurement is performed by a suite of standardized tests with normalized exercises to test general or specialized abilities. Out of these results the rate for the intelligence, the intelligence quotient, is calculated.

A test in psychological sense is a method of acquiring a sample of a person's behavior under controlled conditions ([Walsh85]). This definition can easily be adapted to fit to programs by changing "person" with "program", "module", "procedure", etc. Tests are an important part of the assessment process, but they do not define the process. Tests can only sample a portion or segment of the totality the capabilities and characteristics, but we face the assumption that the observations we get from test have to be generalized to the overall behavior.

---

1 Authors permanent address: Erich Schikuta, Institute of Applied Computer Science, Dept. of Data Engineering, University of Vienna, Rathausstr. 19/4, A-1010, Vienna, Austria

2. Problems of static measures

Measures are usually defined as an assignment of numbers to objects to describe certain properties or attributes ([Fenton90]) and to compare objects to each other. All conventionally published measurement or comparison methods have the disadvantage to check the static properties of the software systems only, e.g. number of procedure calls, number and size of input/output variables etc. We are convinced that the static (syntactical) situation of a software program reflects only inaccurate the situation of the dynamic behavior of the system, like actual number and type of procedure calls, size of the actual transferred information etc. Only dynamic characteristics present us a real picture about the coupling in software system.

Static software measures are obviously defined on code. A problem for a general and objective approach in the measurement theory are the employment of different programming languages with different programming paradigms, the personal style of the programmer, the project regulations, the underlying hardware, etc. This leads to the situation that programs are difficult to compare objectively. We want to prove this statements by a few examples.

In [Henry81] with the analysis of the Unix program code the 'u' module in Unix was excluded of the static measurement calculations, because it didn't fit in the presented FAN-IN/FAN-OUT method. It delivered impressive numbers in the static analysis, but didn't reflect the program characteristics correctly. Because of the special task of the u module in the Unix system, to provide the error handling, the coupling degree was tremendous. In practice this wasn't reflected by the number of error corrections (an external attribute, see next section) performed on that module, which was used to prove the presented static measure. The execution probability of this module was less than the probability of the other modules. This fact is easily recognizable with a dynamic analysis, but is very hard to find (after an excessive logical or semantic analysis only) by static measures.

Static measures are influenced by the written program code. Therefore rather intuitive rules for static evaluation (see [Conte86]) are necessary. For example calculating the well known Halstead [Halstead77] measures variables have to be carefully exclude from the count that are not used in the actual but defined for a future program execution. These variables can not be counted, because they do not affect the operational characteristics of the program or the difficulty of the development.

Problems arise also in analyzing programs written in programming languages, which support extensively the use of addresses to reference variables or better memory locations, like C and its derivations. Static analysis fails in many cases to determine, which defined variable is accessed by a address actually. The analyzing process is desperately mixed up with unknown and untyped address references. Interestingly the same problem restricts (or prohibits) the use of C as a language for the programming of parallel computer systems. Parallelizing compilers can not resolve dependencies in C programs because of the unknown address references. This leads to the use of FORTRAN as the general language for these application type, which uses static memory allocation.

3. Dynamic approach

We recommend a dynamic approach for the measurement of the coupling of software systems. The behavior has to be analyzed during the execution of the program and out of the behavior of a number of characteristic values a metrics for the coupling degree can be derived.

Fenton defines in [Fenton90] three basic classes of entities of interest for software measurement. These classes are processes, products and resources. We argue to add to these classes a new class of dynamic behavior. This is based on the fact that all measurement methods serve as a mean to understand internal and external behavior of the software product.

External attributes are derived attributes of the module, which are dependent on one or more entities in addition to the module ([Fenton90]). This attribute form is also known as psychological complexity measure ([Melton90]) and quantifies reliability, understandability, maintainability, estimated production costs, etc. Internal attributes are static attributes of the module without dependence to any other entity. They comprise the length, the syntactic correctness, the modularity, the reuse, etc. of a module.

The aim of measurement theory is to derive a measure for the internal attributes, which accurately describes external attributes.

The validation of static measures is a difficult. Only construct validity ([Kluwe91], [Sternberg82]) seems appropriate, where the measurement characteristics have been chosen on the basis of 'some' theory. Dynamic measures give the possibility to recognize interpendences between internal and external attributes which are not known or suspected in advance.

We propose the use of dynamic measures as describing attributes. These dynamic attributes give, in our opinion, a more realistic picture of the actual module behavior and reflect purer the effect on the external attributes.

This leads to the conclusion that dynamic interconnections describe the coupling in a system better than their static counterparts. Static interconnections are described by the programming code only, but dynamic interconnections by the actual run time behavior of the code. Troy and Zweben [Troy81] specify different metrics, which are derivable from the structure of a module, and state that all contribute to coupling. This statement is hereby extended that the dynamic counts also describe the level of coupling.

The intuitive 2 axioms for coupling ([Fenton90]), which give an empirical relation for connectivity of a module, have to be extended by additional axioms describing dynamic connectivity. These can be formulated by the following 2 additional axioms:

Axiom 3

> If an additional *dynamic* interconnection is added to a module, than its level of coupling is increased .

Axiom 4

> If a module C is added to a system consisting of 2 modules A and B and the pairwise *dynamic* coupling between A and C and B and C is the equal to the *dynamic* coupling between A and B than the global *dynamic* coupling doesn't change.

The dynamic axioms are basically the same as the original one, but extend the meaning to dynamic connections.

3.1. Standardization

Within the dynamic approach measures can not be used directly. A problem of dynamic measures is that they are heavily depend on the time dimension. Obviously if a test runs longer than another, the values will be greater. Therefore the calculated numbers of dynamic measures have to be treated as *raw scores* and must be adapted accordingly. The raw score must be converted into a measure of relative standings in comparison to a normative group. This leads to definition of a *norm*, which standardizes the raw scores correspondingly to a basis and gives the possibility to compare the standardized scores. A number of possibilities can be seen to standardize calculated measures.

3.1.1. Ratio of the measure to the execution time

Because of the dependence of the measure value to the time dimension, it is the easiest way to build the ratio of the value and the executed test time to get comparable numbers. We call these measures time-standardized m_{ts}, where

$$m_{ts} = \frac{m}{t}$$

m is the measure and t is the executed test time.

This measure has a limitation, if you analyze programs with a strong I/O component or which access shared resources. These programs show a long wait time, which is obviously influencing the time-standardized measure value. This problem can be partially solved if the system allows to break down the execution times regarding to the process state.

Further it is not possible to compare results of different system architectures. The numbers are heavily influenced by the hardware properties (like processing power, transfer rate of the disk system, etc.) or the system software (multitasking facility, swaping- or paging policy).

This leads to the conclusion that execution time standardized measures are limited to a small set of applicable situations, like comparing calculation intensive applications on a single machine only.

3.1.2. Ratio of the measure to the number of executed statements

A standardization can be achieved also by dividing the measure by the number of executed statements. The number of executed statements is obviously directly correlated with the execution time of the program. This approach is similar to the counting of lines of code with many static measures. Problems arise with the expressive power of different programming languages and the style of the programmer. So commonly a line of code is defined as any line of program text that is not a comment or blank line , regardless of the number of statements or fragments of statements on the line. This specifically include all lines containing program header, declarations, and executable and non-executable statements ([Conte86]).

Dynamic measurements have the advantage that they can use the compiled form of the program and are therefore freed from specific programming language features. The personal style of the programmer is kept and can be judged by the analysis. The compiled line of code can than be defined as the executable statement of the object code. This number gives a better basis for calculating the ratio than the source code. We call a standardization based on the executed lines lines-standardized and define it by m_{ls}, where

$$m_{ls} = \frac{m}{l}$$

m is the measure and l is the number of executed lines of code.

A disadvantage can be the differences of compilers, which (can) produce different codes for the same source program and different hardware (processor) architectures, which constitutes different executable statement formats. This problem can only be avoided by comparing the results of modules only, which are compiled by the same compiler, using the same compiler options and running on the same hardware type.

The ratio of measures to executed statement gives a valid instrument, but is suited to a certain group of problems only.

3.1.3. Standardization according to the measure distribution

Distribution scores express the distance of the individual score from the mean of the distribution of the measure values of the group of observed program units in standard deviation units. The group can be defined by all procedures of a module or all modules of program. This standardization procedure is obviously only feasible for an acceptable large group. Which group size is acceptable is in the responsibility of the analyzer, but at least certain basic statistical rules have to be adhered (e.g. for the use of a statistical test).

Distribution scores can easily be calculated by z values, which are defined by

$$z = \frac{m - \mu}{s}$$

m is the measure, μ is the mean of the measure values of the respective group and s is the standard deviation of this group

Further we can propose a underlying normal deviation and can normalize the standard score and transform it to fit a normal distribution. This solves the problem in comparing the measure values to different distributed samples (the shapes of the measure distribution is different)

This approach allows us to calculate a type of standard score similar to "Stanines" ([Walsh85]), which can be used in the calculation of coupling numbers. (We could transform the values to the interval 0 to 1, instead of the original 1 to 9).

After these general considerations of calculating comparable dynamic measure values, we want to give a group of actual values, which can be used in the dynamic analysis process.

3.2. Measures

The dynamic measures can be defined analogous to the commonly known measures found in the literature. A few examples are give by

- FAN-IN/FAN-OUT ([Henry81])
- length of the information stream (see Henry-Kafura metric [Henry81]
- storage locations references
    - calls by name (binding of names to different (and changing) storage areas)
    - calls by reference (references to variables outside the procedure storage area)

- calls by value (copying of values to the procedure stack)

... (and many more)

3.2.1. Dynamic measures for the interconnection between modules

We give two dynamic measures for the coupling of modules. The first is based on the FAN-IN/FAN-OUT measure of Henry and Kafura ([Henry81]), the second on the pairwise coupling measure M of Fenton ([Fenton90]). To express the standardization, we use the transformation function t(m). It is necessarily in this context not decided, which standardization is applied.

The presented approach is easily extendible to all dynamic measures.

3.2.1.1. Dynamic FAN-IN/ FAN-OUT

Henry and Kafura defined in [Henry81] the fan-in of a module A as the number of local flows into module A plus the number of data structures from which module A retrieves information. Consequently the fan-out of a module A is defined by the number of local flows from module A plus the number of data structures which module A updates. The measure for the coupling is described by a family of formulas as

$$\text{fan-in} * \text{fan-out}$$

$$(\text{fan-in} * \text{fan-out})^2$$

$$S * (\text{fan-in} * \text{fan-out})^2,$$

where S is the size of the module

According to the given definitions we can define a dynamic fan-in and dynamic fan-out.

The dynamic fan-in (fan-in_{dyn}) is the number of memory location references which are read by the module during its execution.

The dynamic fan-out (fan-out_{dyn}) is the number of memory location references which are written by the module during its execution.

We can now easily define a family of dynamic $\text{FAN-IN/FAN-OUT}_{dyn}$ formulas as follows,

$$t(\text{fan-in}_{dyn} * \text{fan-out}_{dyn})$$

$$t((\text{fan-in}_{dyn} * \text{fan-out}_{dyn})^2)$$

$$t(S * (\text{fan-in}_{dyn} * \text{fan-out}_{dyn})^2)$$

where t is the standardization transformation and S is the size of the executable code of the module.

3.2.1.2. Dynamic pairwise coupling measure of Fenton

In [Fenton90] a measure M(x, y) for the pairwise coupling of modules x and y is defined, which based on the rank of the maximum coupling type. These coupling types are defined by six different classes of coupling, which are given by the following table

empirical order	type	description of interconnection of module x to y
0	no coupling	totally independent
1	data coupling	x and y communicate via parameters

2	stamp coupling	x and y accept the same record type as parameter
3	control coupling	x passes control parameter to y
4	common coupling	x refers to the same global data as y
5	content coupling	x refers to the inside of y

A natural order is defined on the types of the coupling, i.e. when M(a, b) = 4 and M(b. c) = 3, then M(A, b) > M(b, c). For a more information see [Fenton90].

The static measure M(x, y) for the coupling of 2 modules x and y is given by the formula

$$M(x,y) = i + \frac{n}{n+1}$$

where i is the greatest coupling type of x and y (e.g. i = 4 if x and y show common coupling) and n is the count of static interconnections between x and y.

This measure can easily dynamized by using the actual dynamic interconnection type i_{dyn} and the dynamic interconnections count n_{dyn}, i.e. the dynamic references of the specific type, as a count for n. We have to consider (similar to the dynamic FAN-IN/FAN-OUT) the standardization of n_{dyn} by a function $t(n_{dyn})$. The new dynamic measure for the coupling can be defined by the formula

$$M_{dyn}(x,y) = i_{dyn} + \frac{t(n_{dyn})}{t(n_{dyn}+1)}$$

Both FAN-IN/FAN-OUT_{dyn} and the M_{dyn} coefficient preserve the properties of a valid measure according to [Melton90], where the properties of relations are used to describe orderings. These properties are reflexivity, antisymmetry and transitivity and can be defined by a given set S and a relation R on S, where R is

- reflective if (x, x) ∈ S;
- antisymmetric if, whenever (x, y) ∈ R and (y, x) ∈ R, x = y;
- transitive if, whenever (x, y) ∈ R and (y, z) ∈ R, (x, z) ∈ R

These definitions are used to describe orderings, where given a set S and a relation R on S

- R is a preorder and (S, R) is a preordered set if R is reflexive and transitive;
- R is a partial order and (S, R) is a partially ordered set if R is reflexive, antisymmetric and transitive;
- R is linear or a total order and (S, R) a linearly or totally ordered set if R is a partial order and if, for each pair of elements x and y in S, either (x, y) ∈ R or (y, x) ∈ R.

For more information and adjoining examples see [Melton90].

Obviously the FAN-IN/FAN-OUT_{dyn} and M_{dyn} measure show the same property as the original measures and are preorders.

4. Testenvironment

4.1. Program execution profile

Basically two different types of program execution profiles can be distinguished,

- artificial profile and
- reality profile

An artificial profile reflects a test environment, which observes the characteristics and the behavior of the module in specialized cases, which are designed by the analysis process. This are situations, where a comprehensive test is done to check the overall functionality of the system. It also covers exceptional cases, which are encountered by the modules only occasionally or never during the conventional execution cycle. These can be errors, peakload or overload situations. Artificial profiles are normally a by-product of the software development process for the validation of the product, like module test, function test, etc.

The reality profile simulates the practical usage of the system and is determined by the expected actions of the user and the forecasted system load numbers. In other words, it reflects the actual situation in which the program is normally executed.

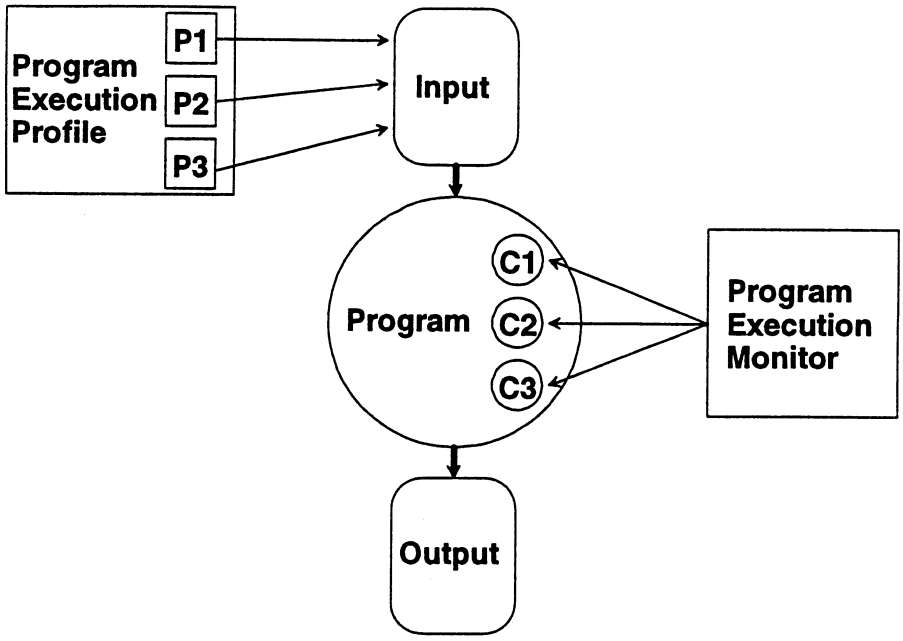
This an analogy to intelligence tests for general or specialized abilities in the psychology ([Walsh85]).

For software systems two additional factors seem appropriate to examine, program load and locality. The program load is described by the number of executed and completed program tasks per time unit. Program locality depicts the focus of the program execution to single modules. For example, with an commercial accounting system the program load is reflected by the workload of the system (number of booking entries) and the locality by the system tasks (entering of entries, balancing of books, etc.).

4.2. Monitor system

To determine the system characteristics a special monitor program has to be established similar to the systems known from system profiling or debugging. This monitor program runs virtually parallel to the software system, supervises the behavior during execution and calculates the characteristics at defined time intervals. Therefore it is possible to get different characteristics views for varied execution profiles, given by program locality or workload.

The following figure pictures the described framework.



Dynamic measurement framework

P1 to P3 denotes different execution profiles and C1 to C3 different dynamic program characteristics.

This monitoring process should be part of the developing environment too. This allows an incremental check of the interesting measures during the development process. The design of the modules can be checked step by step and arising problems can be detected by the measures in an early stage. This guarantees an efficient and controlled development process. By the automatic employment of this incremental supervising system a high level of objectivity can be reached. The common problem of the subjective and therefore often inefficient self-control of the programmer is diminished and also the costly expense of a 'third' objective controller is solved.

It can be concluded easily that by the integration of an automated monitoring system into the software development environment the arising development costs can be decreased substantially.

5. Summary and Conclusion

We gave a model for a dynamic approach to measure the coupling of software systems. The conventional used measurement systems are static oriented and give only measures for syntactical program analysis. This leads to inaccurate results, which reflect the dynamic software behavior only incompletely. With our dynamic approach we calculate dynamic measures dependent on varying execution profiles. This is performed by a monitor system, which can be integrated into the development system.

We think that dynamic measure counts have not to stand alone, but have to be seen together with their static counterparts or other measures. But we are convinced that only with the dynamic component an overall view of the programming module in question can be given.

An additional advantage of dynamic measures is their capability to evaluate the reliability of a programming system, which describes the systems behavior dependent on a time factor (e.g.

Mean Time Between Failure) or a workload (e.g. threshing behavior dependent on the workload). A discussion of this approach is beyond the scope of this paper and a topic for further research.

6. Acknowledgment

I would like to thank the CRPC and its researchers for providing a stimulating and supportive atmosphere that contributed to this work.

This research was in part supported by the grant J0742-PHY of the Austrian FWF.

7. References

[Card90] Card, D.N., *Measuring software design quality*, Prentice Hall, 1990

[Conte86] Conte S.D., Dunsmore H.E., Shen V.Y., *Software engineering metrics and models*, Benjamin/Cummings Publ., 1986

[Fenton90] Fenton N., Melton A., "Deriving structurally based software measures", *Journal of Systems and Software*, 12, pp. 177-187, 1990

[Halstead77] Halstead M.H., *Elements of software science*, Elsevier, 1977

[Henry81] Henry S., Kafura D., "Software structure metrics based on information flow", *IEEE Transactions on Software Engineering*, 7, 5, pp. 510-518, 1981

[Kluwe91] Kluwe R.H., Misiak C., Haider H., "The control of complex systems and performance in intelligence tests", In H.A.H. Rowe (Ed.), *Intelligence: Reconceptualization and Measurement*, Lawrence Erlbaum Ass., 1991

[Melton90] Melton A.C., Gustafson D.A., Bieman J.M., Baker A.L., "A mathematical perspective for software measures research", *Software Engineering Journal*, pp. 246-254, Sept. 1990

[Myer76] Myers G., *Composite/Structured Design*, Van Nostrand Reinhold Company, 1976

[Sternberg82] Sternberg R., "Reasoning, problem solving, and intelligence", In R.J. Sternberg (Ed.) *Handbook of Human Intelligence*, Cambridge University Press, 1982

[Stevens74] Stevens W.P., Myers G.J., Connstantine L.L., "Structured Design", *IBM Systems Journal*, 2, 1974

[Troy81] Troy D.A., Zweben S.H., "Measuring the quality of structured design", *Journal of Systems and Software* 2, 113-120, 1981

[Walsh85] Walsh W.B., Betz N.E., *Test and Assessment*, Prentice Hall, 1985

[Yourdon79] Yourdon E., Constantine L.L., *Structured Design*, Prentice Hall, 1979