Measuring the Entropy of Large Software Systems

Greg Snider
HP Laboratories Palo Alto
HPL-2001-221
September 10th , 2001*

entropy, software systems, structure, metric

How does one measure a large software system to determine if it is "well-structured"? This report proposes a metric for doing just that, based on the concept of entropy from information theory. A tool that automatically extracts the metric from source code was built and used to compare two large software systems (each about 500,00 lines of source code): a commercial application that has existed and been heavily modified for several decades; and a recent rewrite of the same system that aimed at producing a well-structured system. The rewritten system was shown to have much lower entropy in each of its subsystems, compared to the legacy system, as well as much lower entropy overall.

1 Problem

A common goal of software architecture is to maximize cohesion within modules and minimize coupling between modules—systems meeting those goals are said to be well-structured. The problem is: how can you measure the degree of conformance of a large software system to the principles of maximal cohesion and minimal coupling? The size of large software systems makes manual evaluation impractical, and subjective evaluations are vulnerable to bias [1].

An automatic tool for performing such measurements has at least three areas of application in building and maintaining large software systems:

- Comparisons—if one if forced to choose between two implementations of a software system with essentially the same functionality, choosing the better structured one would probably provide lower costs of maintenance and extension.
- Evolution—as large software systems grow, their structure is subject to degradation. Measurement provides and means for detecting such degradation so that countermeasures can be initiated.
- Restructuring—a measurement tool can provide restructuring guidance to minimize ad hoc decision making.

2 Background—Software Metrics

Software metrics are indirect measurements of selected “quality” attributes of a software product or process that are used to either (1) estimate the presence or value of the desired attribute, which is difficult to measure directly, from real-world inputs that are relatively easy to measure; or (2) make predictions of the future value of the desired attribute based on inputs and parameters that can be specified in the present. The expectation is that metrics can provide useful feedback to software designers as to the impact of decisions made during coding, design, architecture, or specification; without such feedback, many decisions must be ad hoc.

As shown in figure 1, a software metric requires [2]:

(1) A specification of raw inputs to be measured (e.g. lines of source code, historical defect rates).

(2) A model that maps inputs and parameters to the metric (e.g. defect rate = f(lines of source, historical defect rate)).

(3) A specification of parameters for adjusting the model.

(4) A specification of the metric’s meaning—that which is to be measured or predicted (e.g. the defect rate of a software system).

(5) A means of comparing the metric with other measurables to empirically validate the metric's usefulness. If this cannot be done, the metric is "metaphysical."

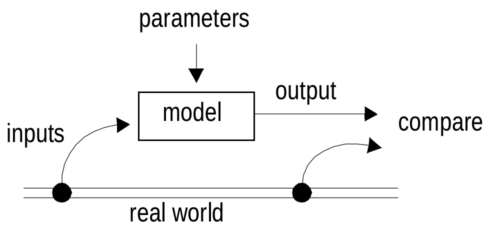

FIGURE 1. A metric requires (1) inputs measured from the "real world", (2) a model, (3) parameters to adjust the model, (4) a predictive output, and (5) a means of validating the output by comparing it with measurement from the real world..

The track record of software metrics has been generally poor [3,4,5]—Sheppard and Ince [2] propose three reasons for this:

- A metric lacks clear definitions of terms. For example, "complexity" and "quality" are too ill-defined to be useful.
- The metric's theoretical basis is overly ambitious, inconsistent or unconvincing.
- The metric lacks empirical validation, has conflicting empirical results, or makes predictions no more powerful than those produced by the infamous "lines of code" metric (which is generally used as a "null hypothesis" in evaluating other metrics).

3 Prior Work

Popular metrics which are suspect or largely discredited for one or more of the reasons mentioned in the previous section include Halstead's Software Science [6], McCabe's cyclomatic complexity [7], Henry's information flow [8], Emerson's cohesion [9], Yau and Collofello's stability [10], and others. Metrics with better acceptance include the function point metric [11] (which is primarily aimed at predicting development effort from specifications and measuring productivity), and metrics which focus on intermodular connections [12].

These results suggest that expectations for software metrics must be tempered: a precision tool capable of measuring desired quality properties is not likely to be created in the near future (and may be fundamentally impossible[13]). The most that can be expected is a coarse, automated tool that can provide feedback useful to designers. Software is still largely an art, not a rigorous engineering discipline, and ultimately one is forced to rely on designers' experience, talent and judgement.

4 Measuring Structure

We wish to construct a software tool which measures the degree of conformance that a software system has to the architectural principles of high intra-module cohesion and low inter-module coupling. The input to the model is the source code of the system to be measured. The output is a numeric measure of the degree of conformance.

The proposed metric is based on the concept of entropy from information theory [14]. Although entropy is capable of providing a theoretical lower bound on the amount of information contained within “messages,” it is not always computable, and even when it is, it does not prescribe a scheme for achieving that lower bound. Nevertheless, the concept forms the core of the metric: the entropy of system structure is not computed directly, but estimated based on encoding schemes that follow the spirit of Huffman codes.

The following assumptions are used in the construction of the metric:

(1) Since engineers work with source code when modifying a system, we are interested in the structure of the application at the lexical level.

(2) We are more interested in analyzing global relationships than local ones.

(3) The more dependencies a module has on other parts of the system, the harder it is to modify.

(4) “Remote” dependencies are more expensive (in terms of comprehensibility) than “local” dependencies (restatement of cohesion and coupling principle).

The following subsections construct the five parts of the metric as shown in figure 1. We assume for simplicity of explanation that the software to be evaluated is written in the C language, but the techniques described here may be applied to software written in any language.

4.1 Inputs

The primary lexical abstraction in C is the *symbol* (also called *name* or *identifier*). A symbol is a character string that has the properties of: *scope* (global, file, parameter or block) which partially defines the namespace for that symbol; *type*, which is constructed from the built in types and the pointer, struct, union, array and enum operators; and *storage class* (auto, extern, register, static, typedef). Following the assumptions above, we restrict our focus to dependency relationships of symbols with global and file scope. These will be referred to as *global symbols* and include:

- function names.
- global variables[1].
- global struct and union type names.
- fields of global structs and unions.

Since we are interested in the large-scale structure of a software system, symbols that are strictly local in nature are excluded:

- parameter names.
- variables local to a function.

1.A global array is treated like a global scalar in that the index is ignored.

4.2 Model

The inputs of the metric are transformed to a model for analysis. The model proposed is a directed graph composed of two different types of nodes—*leaf nodes* and *interior nodes*—and two different types of edges—*structural edges* and *dependency edges*.

A *leaf node* corresponds to a global symbol, or set of global symbols, in the source code. Function names, global variables, and global struct names each get a leaf node, while fields are collected with the leaf node of the enclosing struct name (since fields are already collected lexically, there is little to be gained by representing them separately).

An *interior node* corresponds to either: (1) an aggregate of leaf nodes (i.e. a *.c* file or *.h* file); or (2) an aggregate of other interior nodes (such as a directory, or a subset of a directory distinguished by a naming convention).

Structural edges, attached to interior and leaf nodes, create a tree that corresponds to the directory and file structure (and possibly file naming conventions) of the source. Note that a structural edge may connect two interior nodes, or an interior node with a leaf node, but may never connect two leaf nodes. Figure 2 shows an example of a model with interior nodes, leaf nodes and structural edges (but no dependency edges).

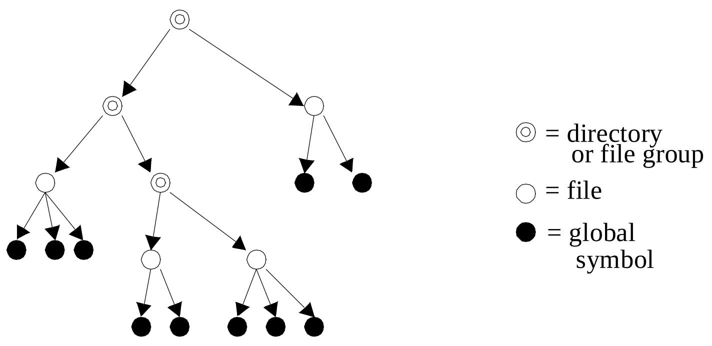

FIGURE 2. Structural edges relating interior and leaf nodes. Leaf nodes are represented by filled circles.

A *dependency edge* interconnects two leaf nodes and represents a dependency that one symbol has on another. Two types of dependencies [15] will be considered:

- *Service dependencies*—if a function, A, invokes another function, B, and A depends upon B's correct implementation, then A has a service dependency on B. This is represented by a directed edge from A to B. For simplicity, all function calls will be assumed to represent service dependencies (although there are cases where this is not true), and indirect invocations (e.g. traps, interrupts) will be ignored.
- *Data dependencies*—if a function references a non-function global symbol, it has a data dependency on it. Global variables of type struct (or pointer to struct) have a data dependency on the struct symbol. Functions with a return type of struct (or pointer to struct) or with parameters of type struct (or pointer to struct) parameters have a data dependency on the struct symbol.

Other dependencies, such as environmental dependencies, are not considered because of the difficulty of extracting them from source code.

Figure 3 shows an example of service dependency edges that result from A calling B, and B calling C and D in the source code.

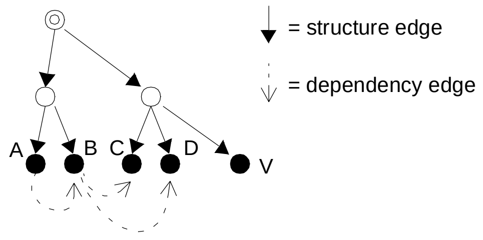

FIGURE 3. Service dependencies: A calls B, B call C and D

Figure 4 shows an example of data dependency edges that result from a shared variable, V, referenced by functions A, B, and C.

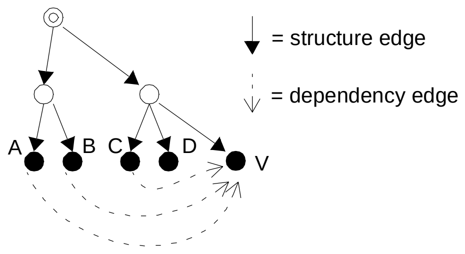

FIGURE 4. Data dependencies. A, B, and C all reference global variable V.

A larger example is shown in figure 5.

foo.h:
```
struct foo {struct foo *next; int value;};
```

foo.c:
```
#include “foo.h”
set_value(struct foo *foo, int val) {foo->value = val;}
```

bar.c:
```
#include “foo.h”
struct foo bar;
main() {set_value(&bar, 5);};
```

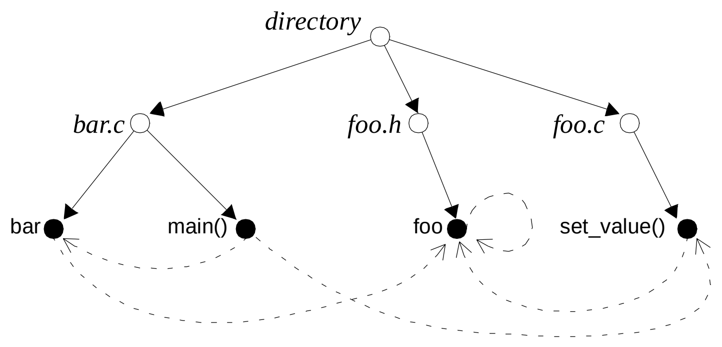

FIGURE 5. Structure and dependency graph for a small program residing in three source files in a single directory.

4.3 Output

Information theory deals with the efficient transmission of messages from a sender to a receiver. The receiver must have some knowledge about the messages to be sent, e.g. whether the messages contain text, graphics, or other information, along with the encoding scheme used to send them. This common knowledge between the sender and receiver, the *context*, is open to negotiation prior to transmission—the receiver must have some knowledge about the messages to be sent; it is the sender’s job to tell the receiver through messages what the receiver *doesn’t* know.

What is the context for the model in this paper? We assume the receiver knows only the following about a software system:

(1) Source tree layout (this corresponds to the *nodes* and *structural edges* of the graph described in the previous section).

(2) The symbols contained within each file (this corresponds to the *leaf nodes* of the graph).

(3) An encoding scheme for describing the dependency edges.

What the receiver desires to learn, through efficiently encoded messages, is the system’s set of service and data dependencies. A *message* is defined to be the precise description of the dependencies a single leaf node (function) has on the rest of the system. The encoding scheme for the messages is derived from the statistics of the software system itself in such a way as to minimize the average length of the messages to be sent[1]. Note that this scheme requires that a message be sent once and only once for each leaf node in the graph representing the system. It will be argued here that the shorter the average message length, the more the system adheres to the objectives of cohesion and coupling.

The structural metric to be derived from the model, call it μ, addresses the following question:

> What is the average number of bits needed to describe the dependencies a function has on the rest of the system?

In other words, what is the average length of a message? Part of the rationale for asking this question follows from assumption 2—the more dependencies a module has on its environment, the more bits are needed to describe those dependencies, and hence the more difficulty there is in understanding that module.

As a simple example, consider a system consisting of F functions (all residing in a single source file) and E dependencies. This will map to a graph with F leaf nodes, 1 interior node (representing the source file), and e dependency edges (figure 6):

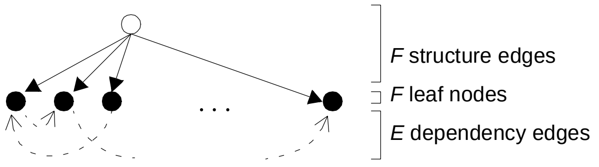

FIGURE 6. Graph of a simple system in a single source file combing structure edges and dependency (service and data) edges.

Assuming that the leaf nodes in this example are numbered from 0 to f-1, it's clear that each edge can be uniquely described by a simple binary encoding of its source and destination nodes:

$$\text{bits per edge} \approx 2 \log_2 F$$

Since there are e/f dependency edges per node on average, the number of bits per node required to specify the dependencies is:

$$\mu = \text{bits per function}_{\text{avg}} \approx 2\,(E/F) \log_2 F$$

So far so good—increasing the number of dependencies increases the value of μ, in line with assumption 2. But the example is a lexically unstructured system—we can lower the value of μ significantly by exploiting the lexical structure of a system (if it exists). The trick is to specify the target of each dependency edge *relative to the tree formed by the structural edges*: since the objective is to minimize the number remote connections relative to the number of local connections, fewer bits may be used to specify the total connectivity.

1. Information theory requires that the message source be ergodic—that its statistics be stationary and independent of initial state. The model here assumes that this is at least plausible for this problem since some natural languages, such as English, have been shown to be approximately ergodic.

First, some definitions:

F:	number of leaf nodes in graph of system
E:	number of dependency edges in graph.
edges(f):	number of dependency edges sourced by leaf f.
dist(f_1,f_2):	distance between leaf f_1 and leaf f_2 (= length of edge (f_1, f_2)). Distance is defined as being the minimum number of interior nodes that must be traversed to trace a path from f_1 to f_2.
p_{src}(e):	probability that a leaf sources e edges.
p_{dist}(d):	probability that two random leaves will be a distance of d apart
p_{edge}(d):	probability that a dependency edge will be of length d
loc(f):	lines of code in leaf function f.
H_{srcs}:	entropy of count of edges sourced by leaf.
H_{edge}:	entropy of specifying edge destination if source is known.
μ:	entropy of message describing a leaf.

Transmitting the dependency edges requires *F* messages, one message for each leaf node. Since the receiver already knows the source structure, the messages can be sent sequentially in, say, lexicographic order, e.g.:

{ leaf(0), leaf(1), leaf(2), ... }

Each message is just a list of the dependency edges sourced by the leaf associated with that message. For example, a message for leaf node f could be of the form:

message(f) = { $edge_{f,1}$, $edge_{f,2}$, $edge_{f,3}$, ... , $edge_{f,n}$ }

An edge can be specified by a (source node, destination node) tuple, but since the messages are sent in the order of the leaves, the source node may be factored out and inferred from the message's position in the sequence, and therefore does not need to be explicitly sent. It is sufficient to encode the number of edges included in the message so that message boundaries may be identified. For example, the message for leaf node f could be encoded as:

message(f) = {edges(f), $dest_1$, $dest_2$, $dest_3$, ... , $dest_n$ }

The entropy of such a message (the average number of bits needed per message) is thus

$$\mu = H_{srcs} + (E/F)\, H_{edge}$$

where

$$H_{srcs} = - \Sigma\, p_{src}(e) \log_2 p_{src}(e)$$

with the summation over e from 0 to the maximum number of edges sourced by any leaf in the graph; and

$$H_{edge} = \Sigma\, p_{edge}(d) \log_2 (F\, p_{dist}(d))$$

with the summation over d from 1 to the maximum path length in the graph.[1] Note that by assumption 3, p_{edge} is not uniform.

4.4 Parameters

One shortcoming of the metric described so far is that is does not take the size of functions into account when computing the system entropy—there is an implicit assumption that functions are small relative to the system as a whole, and that the internal complexity of a function is of no consequence. Unfortunately, this makes it possible to subvert the metric by expanding functions inline in the functions from which they're called. This can be overcome somewhat by refining the specification of a dependency edge's endpoints to include not only the function, but the *line number* within the function as well. This does not change the number of bits needed to specify the destination of an edge, since, in C anyway, a function may only be entered through its first line. The source of an edge, though, would require the additional specification of the line within the source function where the dependence occurs. If we assume that the receiver is informed of the number of lines of code in each function before transmission of the dependency messages, and that all source lines within a function are equally likely to be the source of a dependency, the entropy of an edge would become:

$$H_{edge} = \Sigma\, p_{edge}(d) \log_2 (F\, p_{dist}(d)) + (1/F)\Sigma \log_2(loc(f))$$

where the second term is the average number of bits needed to specify the line within the source function that sources the edge.

4.5 Experimental Result

Large software systems typically comprise multiple source files. These files are individually compiled and linked together to form a single executable. Two tools implementing the algorithms described here have been built—a "dependency compiler" which reads a source file and produces an "object" file which describes the non-local symbols within the source, and the dependencies that each symbol has on other symbols; and a "dependency linker" which takes the "object" files as input to create a single dependency graph that represents the entire program. This dependency graph can then be analyzed to compute the program entropy. The flow of information through this tool is shown in figure 7.

1. The expression $\log_2 (F\, p_{dist}(d))$ is simply the entropy of specifying a destination at distance d where all such destinations at that distance are equally likely. The equation for H_{edge} follows from the independence of the multiple edge description message sources (one source for each distance).

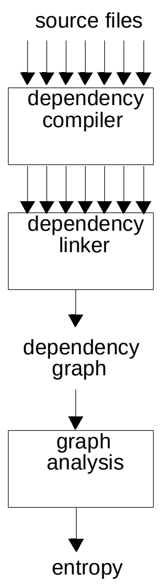

FIGURE 7. Source files of a program are individually fed through a "dependency" compiler to produce "object" files which are linked together to form a dependency graph, which is then analyzed to compute the program entropy.

The entropy tool was used to compare two large software systems, A and B, of nearly the same size (about 500,000 lines of source code) and functionality. Interestingly, both systems produced dependency graph with almost the same number of leaves (15,667 for System A, 15,468 for System B). System A is a large commercial program that has existed for more than two decades, and has been modified by hundreds of engineers over its lifetime in a time-constrained environment. System B is essentially a recent rewrite of the functionality of System A that aimed at producing a well-structured system. Most engineers who were allowed to examine both systems had the subjective impression that system B was better structured than system A.

Each system could be naturally decomposed into seven subsystems. The entropy of each system as a whole was measured, as well as the entropy for each of the seven subsystems. The results, shown in figure 8, agreed well with the subjective impression that System B was better structured (had less entropy) than System A.

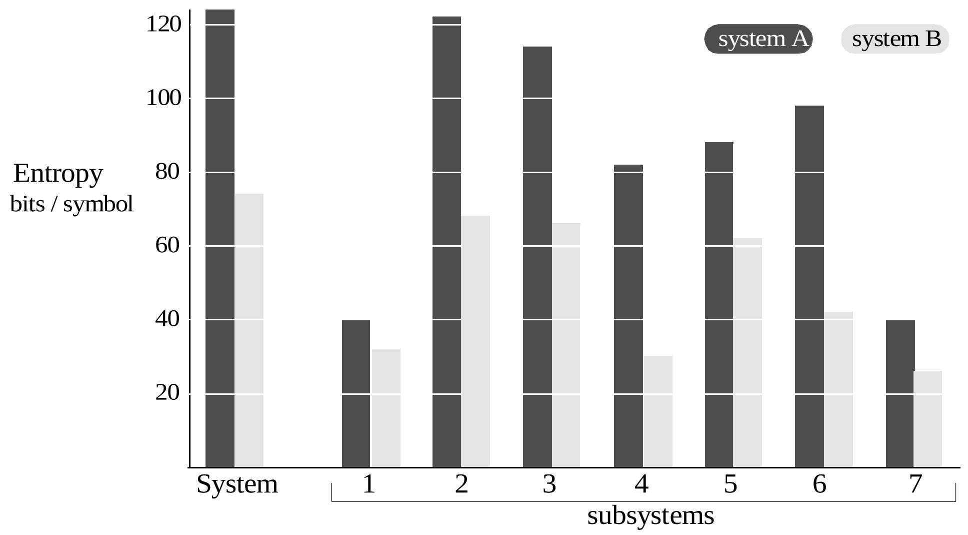

FIGURE 8. Comparison of System A (an old, heavily-modified, legacy system) and System B (a recent rewrite). Each system was composed of the same seven subsytems. The entropy of each subsystem in System B was significantly lower than the corresponding subsystem in legacy System A. System B also had lower entropy as a whole.

References

[1] L. Strigini, “Limiting the dangers of intuitive decision making,” *IEEE Software*, January 1996.

[2] M. Shepperd and D. Ince, “Derivation and Validation of Software Metrics,” Clarendon Press, Oxford, 1993, pages 60-63.

[3] T. Bollinger, “What Can Happen When Metrics Make the Call,” *Transactions IEEE Software*, vol 12, no 1, Jan 1995.

[4] C. Jones, “Software Metrics: Good, Bad, and Missing,” *Computer*, vol 27, no 9, Sept 1994.

[5] M. Shepperd and D. Ince, “Derivation and Validation of Software Metrics,” Clarendon Press, Oxford, 1993, chapters 2, 3, and 8.

[6] M. H. Halstead, “Elements of Software Science” Elsevier-North Holland, 1977.

[7] T. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, vol 2, no 4, 1976.

[8] S. Henry, “Software metrics based on information flow,” *IEEE Transactions on Software Engineering*, vol 7, no5, 1981.

[9] T. Emerson, “A discriminant metric for module cohesion,” *Proceedings of 7th International Conference on Software Engineering*, 1984.

[10] S. Yau and J. Collofello, “Some stability measures for software maintenance,” *IEEE Transactions on Software Engineering*, vol 6, no 6, 1980.

[11] C. Behrens, “Measuring the productivity of computer systems development activities with function points,” *IEEE Transactions on Software Engineering*, vol 9, no 6, 1983.

[12] D. Troy and S. Zweben, “Measuring the quality of structured designs,” *Journal of Systems and Software*, vol 2 no 2, 1981.

[13] N. Fenton, “Software measurement: a necessary scientific basis,” *IEEE Transactions on Software Engineering*, vol 20, no 3, March 1994.

[14] L. Strigini, “Limiting the dangers of intuitive decision making,” *IEEE Software*, January 1996.\

[15] J. Pierce, “An Introduction to Information Theory: Symbols, Signals and Noise,” Dover Publications, 1980.

[16] P. Janson, “Operating Systems: Structures and Mechanisms,” chapter 10, Academic Press, 1985.