SOFTWARE REUSE AND
INFORMATION THEORY BASED METRICS

William R. Torres
Air Force Institute of Technology
Computer Customer Support Center
Wright-Patterson AFB, OH 45424

Mansur H. Samadzadeh
Computer Science Department
219 MS Building
Oklahoma State University
Stillwater, OK 74078

ABSTRACT

The main purpose of this research is to theoretically investigate the effect of reusing software on metrics that are based on the entropy function of communication information theory. R. N. Chanon's Entropy Loading and E. T. Chen's Control Structure Entropy were applied to C and Ada programs obtained from the open literature. Four units of decomposition (statement, component, module, and program) along Chanon's definition of an object were introduced to classify software reuse units. A total of three versions for each of the three programs were considered (optimum reuse, intermediate reuse, and no reuse). The lines of code metric was utilized to quantify the amount of nonreusable code in each of the versions of the programs. Pearson product-moment correlations were computed between the information theory based metrics and the lines of code metric. The results of this study show that there are significant correlations between the information theory based metrics and software reusability.

1. Introduction

The notion of reusing software dates back to the early stages of the history of computing when subroutine libraries were developed [27]. The advantages and limitations of software reuse have been widely publicized ([3], [19], [27]). In particular, it is clear that the extensive reuse of software is likely to reduce software costs during the design and implementation phases (software already exists) and during the validation phase (software has already been checked) [27].

The main purpose of this study is to theoretically investigate the application of software metrics to development environments that employ reuse techniques and principles. More specifically, the effect of reusing software on the metrics that are based on the entropy function of communication information theory is investigated.

The following sections define the metrics used in this study, describe the experimental design, discuss the analysis of the measurements, and summarize and conclude with recommendations for future work.

2. Entropy and the Software Development Process

In communication information theory, information is defined as "what we don't know about what is going to happen next" [25]. Information theory presents entropy as a synonym for information uncertainty or unpredictability. Therefore, entropy can be used as a measure of information.

When an attempt is made to predict the outcome of an event, uncertainty approaches zero as the probability of an outcome approaches unity. On the other hand, uncertainty, as a function of the number of different things that can happen, reaches its maximum value when all the outcomes have the same probability of occurrence (i.e., in the case of equiprobable events). The entropy function, H, which measures information in bits is defined as

\[ H(p_1, p_2, ..., p_n) = \sum_{j=1}^{n} p_j \log_2 (1/p_j) \]

where pj is the probability of occurrence of the jth event.

van Emden [31] defines a system as a set of variables influencing one another. A challenge often faced by software designers when designing a system is managing the complexity resulting from the presence of a large number of variables and the fact that most of them
influence many others. To alleviate this problem, Alexander [2] states that a system composed of a set of variables, \( S \), should be partitioned into subsets or subsystems \( (S_1, S_2, ..., S_j, ..., S_n) \) in such a way that the interactions among subsystems should be minimized with respect to the interactions within a subsystem. This partitioning process allows for the variables in \( S \) to be manipulated freely without any constraints imposed by any of the other subsets.

If a program structure could be decomposed into distinct classes of subsystems, the information contained in the structure can be measured by the entropy metric, \( H([17],[31]) \). More specifically, the entropy metric \( H \) is defined for a system \( S \) as

\[
H(S) = \log_l|X| - \frac{1}{2} \sum_{j=1}^n (|X_j| \log_2 |X_j|)
\]

where \( X, X_1, X_2, ..., X_n \) are the distinct classes of subsystems, \( |X| \) denotes the cardinality of the set \( X \), and \( P_j = |X_j|/|X| \).

### 3. Information Theory Based Metrics

Software metrics are usually categorized as either process metrics or product metrics. Conte et al. [7] define product metrics as measurements of the software product. Product metrics can be collected by analyzing the software using an automated tool. Information theory based metrics can be classified as product metrics. The following subsections discuss the information theory based metrics Entropy Loading [4] and Control Structure Entropy [6].

#### 3.1 Entropy Loading

Section 2 described how we can use the entropy metric to measure the information contained in a system. Using the same basic principles, we can use the entropy loading measure, as described by Chanon [4], "to measure the information shared among subsystems, as opposed to the information contained in each subsystem." He defines interaction as "a shared assumption among two or more objects" where objects are defined as "an identified portion of a program that has net effects." The following is a list of assumptions identified by Chanon.

1. Relationships that must hold prior to the execution of an object in order for its effects to be realized.
2. Data structures or data values.
3. Assumptions about the environment in which an object is executed, such as frequency of usage or order of computation.
4. Assumptions based on mathematical theorems that are relevant to the problems being solved.

Once all the assumptions have been identified, they can be recorded in what Chanon calls an object/assumption table. For a given program, such a table \( T \) is defined for all objects \( I \) and assumptions \( J \) as follows

\[
T(I,J) = \begin{cases} 
1, & \text{if object } I \text{ makes assumption } J \\
0, & \text{otherwise} 
\end{cases}
\]

According to van Emde and as demonstrated by Chanon, the data contained in the object/assumption table characterizes the extent to which collections of objects interact. This data is used in the calculation of a measure which van Emden and Chanon call entropy loading. Entropy loading is defined for a set of rows \( S \) in an object/assumption table at a given time in the development of a system.

Assume that \( S \) is partitioned into subsets \( A \) and \( B \) such that \( A \cap B = \emptyset \) and \( A \cup B = S \). Then \( C(S) \), the entropy loading of \( S \), is given by

\[
C(S) = H(A) + H(B) - H(S)
\]

where \( H(X) \) is given by the entropy metric mentioned in Equation (2).

Chanon [4] demonstrated that programs that possess small entropy loadings also possess properties consistent with the principles of "good" structure stated by Alexander, Dijkstra, Parnas, and Simon.

Based on the aforementioned discussion, entropy loading can be used as a metric (in the tradition of Halstead [8] and McCabe [15]) to determine how "good" a program is, specially when compared to other input/output equivalent programs or different versions of the same program.

#### 3.2 Control Structure Entropy

Chen [6] defines a measure of program control complexity based on an information theoretic viewpoint. Given a strongly-connected proper flow chart which results from structured programming, Chen defines the maximal intersect number, \( MIN \), as "the maximum number of edges which can be intersected by a continuous line drawn such that the line never enters any region, including the external region, more than once." Chen utilizes three basic types of control structures to obtain \( MIN \) from the flow charts: 1) \( SEQUENCE \), 2) \( IF \) \( p \) \( THEN \) \( f \) \( ELSE \) \( g \), and 3) \( WHILE \) \( p \) \( DO \) \( f \).

If a flow chart or graph is not strongly connected, but can be visualized as consisting of more than one strongly-connected subparts connected in series, \( MIN \) can be
obtained from
\[
\text{MIN} = \sum_{i=1}^{n} \text{MIN}_i - 2n + 2
\]  

where \( n \) is the number of strongly-connected subparts and \( \text{MIN}_i \) is the MIN for the \( i \)th strongly-connected subpart. MIN can also be computed analytically from the expression \( \text{MIN} = Z_n + 1 \) where \( Z_n \) is given by the following formula
\[
Z_n = 1 + \sum_{j=2}^{n} \log_2(2p_j + q_j)
\]

where \( n \) is the number of decision symbols on the flow chart or graph, \( q_j \) is the probability that the \( j \)th IF symbol is forming a serial relation with any of its preceding and adjoined IF symbols, and \( p_j = 1 - q_j \). For a given flow chart, \( q_j \) is either 1 or 0 depending on whether or not it is in the specified serial relationship.

Chen describes a programmer to be (in information-theoretic terms) a "channel" who is to handle information from a program specification or problem statement described as the "source." A source of information is characterized by the variety of output symbols that it produces. In this case, the output is a sequence of IF symbols. The other components of a proper flow chart, i.e., function nodes and collecting nodes are considered and ignored because function nodes do not contribute to the branching or nesting structure of a flow graph and collecting nodes are merely converging points for the branches of the corresponding decision points.

\( Z_n \) (see Equation 5) is defined as the control structure entropy of an information source when it emits \( n \) IF symbols. This control structure entropy is a measure of the control variety of a source's output and is correlated with the conceptual complexity of a program. The higher \( Z_n \)'s value, the more complex the program is.

Based on Chen's work, control structure entropy can be used as a quality assessment metric to determine how "complex" a program is, specially when compared to other programs or different versions of the same program.

4. Software Reusability

The notion of software reuse has been around since the early stages of the history of computing when the main motivation for the development of subroutine libraries was software reuse [27]. Software reuse has also been associated with software portability. According to Sommerville, "porting a program to another computer can be considered an example of software reuse although it is possible to reuse a program which is not portable and can only run on a single computer."

4.1 Definitions

Prieto-Díaz and Freeman [19] defines reuse as "the use of previously acquired concepts and objects in a new situation" and reusability as "a measure of the ease with which one can use those previous concepts and objects in the new situation." Consequently, in order for software reusability to be beneficial, the effort to reuse a piece of software (specify, locate, and validate) needs to be smaller than the effort required to develop the software from scratch.

4.2 Advantages and Limitations

The actual cost reduction (if there is any) that can be attained by reusing software is difficult to quantify [27]. The number of applications where code can be reused without any modification whatsoever is small. Nevertheless, cost reduction is not the only advantage of software reusability.

Software reuse increases system reliability ([13], [27]). It is widely accepted that operational use adequately tests software components, and reused components, which have been previously in operational use, should be more reliable than brand new components. As a result of software reuse, programming resource utilization can be improved. The availability of reusable software allows for a better distribution of programming resources since not all the code needed is to be developed anew. Another advantage is the reduction in software development time [27]. Reusing components speeds up system production because both development and validation time should be reduced.

However, as expected, software reusability is not a perfect science and there are limitations that must be kept in mind. This is specially true since some researchers think that the limitations and disadvantages outweigh the advantages ([15], [27]).

The first problem is what to do with a piece of software once it has been determined that it is a candidate for reuse. In a recent article, Tracz presents his "Golden Rules of Reusability." Tracz [30] says that before something can be reused, one needs to find it, know what it does, and know how to reuse it, which go along with the need for means of cataloging, classifying, and retrieving software components. Lubars [12] describes the problem of finding a desired piece of reusable code as the most significant technical barrier to code reusability.
Another problem is the not-developed-here syndrome experienced by some programmers and reflected in some company policies that do not allow non-local programs to be utilized. Still another limitation involves the reusable code itself. Researchers are concerned about how specific or, on the other hand, abstract the code needs to be before reuse pays off. This topic is discussed in the next section on current trends.

4.3 Current Trends

There are two main schools of thought on software reusability. The first one promotes the reuse of ideas and knowledge acquired while developing software, while the other promotes the reuse of particular artifacts and components. Although the second approach is more popular, researchers disagree on how abstract the code needs to be before it may be reused ([13], [19]).

Kernighan [11] presents reuse at the program level utilizing the UNIX® pipe. On the other hand, Matsumoto [14] promotes reusing modules defined at higher levels of abstraction to increase the scope of the reusable code.

5. Design of the Experiment

In this study the intent is to explore theoretically the effects of software reusability on Chanon’s entropy loading metric and Chen’s control structure entropy. But before either calculation can be applied, we need to establish some guidelines for identification of the “program parts” that are to be reused. This is needed because a program part is not defined in Chanon’s definition of an object [4], in fact it is left, apparently intentionally, an unspecific and generic concept.

5.1 Reuse Candidates

Even though program decomposition or partitioning is in general language and application dependent, we define four units of program decomposition which are objects that can be considered to be along Chanon’s object definition, are visible to one another, and can be candidates for reuse. These units of decomposition are statement, component, module, and the obvious one-block partition, the program itself. The definitions of the four units of decomposition follow.

The first unit of decomposition is a statement which is the lowest level at which reuse will be considered and is defined as any executable instruction of a program which makes assumptions about, and is “visible” to, any other object in the program. Executable instructions that do not make assumptions are not considered statements, e.g., NEW_LINE in Ada® or printf("\n") in C. These instructions do not depend on any other instruction for their execution.

The unit of the next higher level decomposition or reuse is a component. A component is a collection of one or more contiguous statements having a name and represented by the implementation of a procedure or algorithm. None of the statements in a component are visible to any other object. The only assumptions that a component can make are about the parameters passed to it when invoked or the visibility of other objects and/or global variables. Examples of components are functions in C, and procedures and functions in Ada.

A module is a unit of decomposition or a candidate for reuse above the component level. A module is a collection of components which has a name and can be invoked by another object. Analogously to the component, none of the components inside a module are visible to any other object. The only assumptions that a module can make are about the parameters passed to it when invoked or the visibility of other objects and/or global variables. Examples of modules are procedures and packages in Ada and functions in C.

The highest level of decomposition or reuse is obviously the program itself. An entire program is the highest level at which reuse can occur. A program is defined as a collection of modules. It is an extreme case since it is the only block in the partition.

The four definitions offered above for an object are not supposed to be rigid prescriptive units of reuse. They are merely the easily recognizable milestones along the decomposition spectrum. In other words, a candidate for reuse can consist of a mixture of the above-mentioned units.

5.2 Theoretical Perspective and Limitations

Now that we have defined the possible ways in which the objects in a program can be identified, we need to find a way to relate these objects to program reusability. But before we do that, we need to establish a basis for comparison. Assume that there is a program, S, consisting of n modules, n components, and n statements. Modules are composed of components and components are composed of statements. Consequently, in general, to keep the potential for reuse high, S can be considered as a set of modules plus some components and even some individual statements.
We can study the reusability of a program, S, by assuming that a new program, S', is to be written and that we can identify a set of existing program parts (statements, components, and/or modules) that can be included in S' thus saving the effort of writing them from scratch. In this manner, we can model or simulate the alternative to writing new code which is reusing an "existing" portion of code as it can be obtained "off the shelf."

The basic case can be established by removing the barriers from all modules and components in S, i.e., allowing all of the statements to be visible to one another. In this case, we attempt to simulate a worst case scenario in which the program has been developed from scratch. A case where a programmer was asked to write the entire program without the option of reusing any modules or components.

At the other extreme, we can model an optimum reuse case where the programmer was asked only to write the main program statements and had the opportunity of reusing all the other independent modules and/or components.

Having described the boundary cases for our study of reusability, we can describe a third case which fits inside our reusability spectrum. This is a case in which the programmer is asked to develop the main program statements along with some of the statements for some of the components anew, and has the option of reusing some of the available modules and components. This generic case brings out the problem of determining which of the statements, components, and/or modules are the best candidates to be "reused."

In general, we have a partition, \( \pi \), on a program, S, defined as a collection of disjoint and nonempty subsets of statements in S whose union is S, i.e., \( \pi = \{ B_\alpha \} \), \( \alpha \in I \), where I is an indexing set, such that \( B_\alpha \neq \emptyset \), for all \( \alpha, \beta \in I \), \( B_\alpha \cap B_\beta = \emptyset \) for \( \alpha \neq \beta \), and \( \bigcup \{ B_\alpha \} = S \) where \( \alpha \in I \). We refer to the sets in \( \pi \) as blocks of \( \pi \) [9]. For example, if S is a set consisting of four elements, \( S = \{1,2,3,4\} \), \( \pi \) can be written in eight different ways if the element sequence order is to be preserved, i.e., \( \{\{1\},\{2\},\{3\},\{4\}\} \), \( \{\{1\},\{2\},\{3,4\}\} \), \( \{\{1\},\{2,3\},\{4\}\} \), \( \{\{1\},\{2\},\{3,4\}\} \), \( \{\{1\},\{2\},\{3,4\}\} \), \( \{\{1\},\{2,3\},\{4\}\} \), \( \{\{1\},\{2,3,4\}\} \), and \( \{\{1\},\{2,3\},\{4\}\} \). Obviously, as the size of \( S \) increases, the number of ways in which \( \pi \) can be partitioned, increases exponentially.

Consequently, in a program with a large number of statements, considering all the possible ways in which the statements can be grouped together becomes prohibitively time consuming.

### 5.3 Coupling and Cohesion

To reduce the number of possible combinations of objects, we use the notions of coupling and cohesion as defined by Stevens, Myers, and Constantine [28]. Stevens et al. state that "the fewer and simpler the connections between objects, the easier it is to understand each object without reference to other objects." The complexity of a system is affected not only by the number of connections but also by the degree to which each connection couples any two objects, making them interdependent rather than independent. Thus coupling is reduced when the relationships between objects are minimized or, in Chanon's terms, when the number of assumptions that the objects make is minimized.

One way to minimize coupling is to maximize the relationships among elements within the same object, i.e., obtaining the objects that display the highest cohesiveness. Consequently, we can reduce the number of possible statement combinations by combining into objects the statements that possess the highest cohesiveness, thus resulting in the lowest degree of coupling. Similarly, we can combine into modules the components that possess the highest cohesiveness.

Based on the aforementioned discussion of coupling, we can now identify some realistic combinations of the statements, components, and/or modules in the original program that potentially can be selected for reuse. This will allow us to compute the entropy loading and control structure entropy of the resulting program consisting of the reused as well as original parts.

### 6. Carrying out the Experiment

The main objective of the experiment is to determine if a relation exists between software reusability and the information theory based metrics: entropy loading and control structure entropy. The major problem is finding a mechanism that can help us quantify the notion of software reuse.

As mentioned in Section 4, one of the benefits of software reusability is the reduction in the amount of software that needs to be written when software is available for reuse. Thus intuitively, for a given software system, an inverse relation exists between software that is reused and the amount of software to be developed anew. The amount of software reused can be correlated with the amount of code that needs to be developed from scratch for a given program. The more code that is available for reuse, the
less new code is needed for the new program.

For this study, the lines of code metric\(^7\) was used to quantify the amount of code that needs to be developed from scratch. The lines of code metric was not applied to the "reused" segments of code since they are not considered as part of the effort of writing the new program.

6.1 Experiment Operation

The next two subsections explain the software tools that were developed and used to collect the measurements.

6.1.1 Programs Developed to Collect the Data. A total of three programs in C were developed on a VAX 11/785 running ULTRIX\(^8\) (see [29] for program listings). The first program was developed to compute the entropy loading of a collection of objects in a given program based on the set of assumptions made by the objects (see Section 3.1). In addition to the entropy loading, this program also computes the average object entropy, system entropy \(H(S)\), and the ratio of entropy loading to the total number of objects. No program was developed to extract the assumptions made by the objects in the programs. All assumptions were manually extracted from the programs based on Chanon's work and are listed in [29].

The other two programs were developed to compute the lines of code metric for the C and Ada programs used in the experiment. Both programs compute the lines of code metric as defined by Conte et al. [7] and go one step further. The lines of code was partitioned into three categories for the C programs, namely, declaration lines, non-declaration lines, and brace lines (i.e., "{" or "}") and no C statements). Ada lines of code were partitioned into declaration and non-declaration lines only. Declaration lines were extracted to investigate if there is any relation between them and the information contained in the objects (i.e., average object entropy and system entropy). Additionally, a relation is expected between the number of brace lines in C programs and \(MIN\) since they are expected to correlate well with the depth of nesting in programs.

6.1.2 Data Collection. A total of three versions for each of the programs included in the study were considered. These versions follow the guidelines for reuse established earlier (see Sections 5.1 and 5.2) as follows: version 1, optimum reuse; version 2, intermediate reuse; and version 3, no reuse. All metrics were applied to all three versions of the programs.

All measurements, with the exception of the control structure entropy, were collected on the VAX 11/785 running ULTRIX. Control structure entropy, \(Z_{con}\), was the only metric that was not computed using a program. This metric was computed using the Maximal Intersect Number charts in [29] and the equation

\[ Z_{con} = MIN - 1 \]

where \(MIN\) is the Maximal Intersect Number determined from the charts as demonstrated by Chen [6].

### TABLE I
<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>Application</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastfind</td>
<td>C</td>
<td>string processing</td>
<td>[16]</td>
</tr>
<tr>
<td>mail</td>
<td>C</td>
<td>database</td>
<td>[23]</td>
</tr>
<tr>
<td>editor</td>
<td>C</td>
<td>string processing</td>
<td>[23]</td>
</tr>
<tr>
<td>int_list</td>
<td>Ada</td>
<td>string processing</td>
<td>[26]</td>
</tr>
<tr>
<td>calc</td>
<td>Ada</td>
<td>numeric</td>
<td>[18]</td>
</tr>
<tr>
<td>address</td>
<td>Ada</td>
<td>database</td>
<td>[18]</td>
</tr>
</tbody>
</table>

6.2 Programs Studied in the Experiment

A total of six programs obtained from the open literature were studied in this experiment (see TABLE I and [29]). Three of the programs were written in C [10] and the other three in Ada [1]. A noticeable difference between the C and Ada programs studied was the number of compilation units. All C programs were compiled as single units, while all Ada programs had two or more compilation units per program.

7. Analysis of the Measurements

All data analysis was done on a VAX 8550 running VMS\(^9\) using SAS ([21], [22]). Standard statistical methods described by Conte et al. [7] were used.

The sample sizes for this study were not arrived at statistically; rather, three correct programs written in each language found in the open literature were used. The three Ada programs have 137, 168, and 493 nonblank, noncomment lines of code and the three C programs have 138, 272, and 286 nonblank, noncomment lines of code. Each of the three versions for each of the programs (optimum, intermediate, and no reuse cases) were considered as separate cases.

Pearson product-moment correlations were computed.
TABLE I
METRICS EVALUATED

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Lines of Code</th>
<th>Con</th>
<th>Str</th>
<th>Obj</th>
<th>Average</th>
<th>SEL to OT</th>
<th>System</th>
<th>System Entropy Loading</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Dec</td>
<td>NDe</td>
<td>Bra</td>
<td>Com</td>
<td>Bla</td>
<td>Tot</td>
<td>Ent</td>
</tr>
<tr>
<td>fastfind1</td>
<td>63</td>
<td>15</td>
<td>36</td>
<td>12</td>
<td>42</td>
<td>8</td>
<td>113</td>
<td>5</td>
</tr>
<tr>
<td>fastfind2</td>
<td>83</td>
<td>16</td>
<td>49</td>
<td>18</td>
<td>50</td>
<td>11</td>
<td>144</td>
<td>8</td>
</tr>
<tr>
<td>fastfind3</td>
<td>138</td>
<td>20</td>
<td>84</td>
<td>34</td>
<td>62</td>
<td>24</td>
<td>224</td>
<td>13</td>
</tr>
<tr>
<td>mail1</td>
<td>38</td>
<td>6</td>
<td>26</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>53</td>
<td>7</td>
</tr>
<tr>
<td>mail2</td>
<td>89</td>
<td>9</td>
<td>60</td>
<td>20</td>
<td>11</td>
<td>14</td>
<td>114</td>
<td>13</td>
</tr>
<tr>
<td>mail3</td>
<td>272</td>
<td>20</td>
<td>184</td>
<td>68</td>
<td>22</td>
<td>37</td>
<td>331</td>
<td>31</td>
</tr>
<tr>
<td>editor1</td>
<td>50</td>
<td>6</td>
<td>38</td>
<td>6</td>
<td>10</td>
<td>4</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>editor2</td>
<td>110</td>
<td>10</td>
<td>79</td>
<td>21</td>
<td>12</td>
<td>10</td>
<td>132</td>
<td>13</td>
</tr>
<tr>
<td>editor3</td>
<td>286</td>
<td>23</td>
<td>197</td>
<td>66</td>
<td>22</td>
<td>31</td>
<td>339</td>
<td>26</td>
</tr>
<tr>
<td>int_list1</td>
<td>29</td>
<td>1</td>
<td>28</td>
<td></td>
<td>10</td>
<td>5</td>
<td>44</td>
<td>1</td>
</tr>
<tr>
<td>int_list2</td>
<td>66</td>
<td>6</td>
<td>60</td>
<td></td>
<td>21</td>
<td>10</td>
<td>97</td>
<td>2</td>
</tr>
<tr>
<td>int_list3</td>
<td>137</td>
<td>11</td>
<td>126</td>
<td></td>
<td>23</td>
<td>17</td>
<td>177</td>
<td>8</td>
</tr>
<tr>
<td>calc1</td>
<td>47</td>
<td>5</td>
<td>42</td>
<td></td>
<td>11</td>
<td>3</td>
<td>61</td>
<td>3</td>
</tr>
<tr>
<td>calc2</td>
<td>67</td>
<td>6</td>
<td>61</td>
<td></td>
<td>11</td>
<td>3</td>
<td>81</td>
<td>16</td>
</tr>
<tr>
<td>calc3</td>
<td>168</td>
<td>22</td>
<td>146</td>
<td></td>
<td>33</td>
<td>10</td>
<td>211</td>
<td>24</td>
</tr>
<tr>
<td>address1</td>
<td>104</td>
<td>27</td>
<td>77</td>
<td></td>
<td>20</td>
<td>8</td>
<td>132</td>
<td>8</td>
</tr>
<tr>
<td>address2</td>
<td>199</td>
<td>37</td>
<td>162</td>
<td></td>
<td>47</td>
<td>19</td>
<td>265</td>
<td>13</td>
</tr>
<tr>
<td>address3</td>
<td>493</td>
<td>58</td>
<td>435</td>
<td></td>
<td>127</td>
<td>59</td>
<td>679</td>
<td>36</td>
</tr>
</tbody>
</table>

Correlations were also computed within the lines of code class and within the entropy loading class. The lines of code class is composed of lines of code (LOC), declaration lines (Dec), non-declaration lines (NDe), brace lines (Bra, only used with C programs), comment lines (Com), and blank lines (Bla). The entropy loading class is composed of the object total (Obj Tot), average object entropy, system entropy loading to object total ratio (SEL to OT Ratio), system entropy, and system entropy loading (see [29] for a list of the measurements obtained). TABLE II contains the metrics evaluated for all programs and TABLE III contains some of the correlations among the metrics. No correlations were computed between the entropy loading metric and the control structure entropy as these correlations were deemed outside of the scope of the experiment. See [29] for a complete list of the correlations.

Correlations using natural logarithm transformations of the measurements were computed. The next two sections analyze the results obtained from the correlations of the metrics.

7.1 Entropy Loading Analysis

Entropy loading, as described in Section 3, is a measure of the information shared among collections of objects as
opposed to the information used inside each collection. Consequently, we can expect a higher entropy loading as the number of collections making assumptions increases (considering each object as a collection only containing itself) since the total amount of information shared increases. This assumption was verified by a strong, positive, and significant correlation (significance of 0.004 or less) between entropy loading and the total number of objects for both C and Ada programs.

The increase in the number of collections was represented by an increase in the amount of code (quantified by the lines of code metric) that needs to be written when the opportunity for reuse is smaller. As expected, a strong, positive, and significant correlation (significance of 0.008 or less) was found between the lines of code and the total number of objects, and between lines of code and entropy loading for both C and Ada programs. In other words, the larger the amount of code needed anew (smaller reuse), the larger the number of objects and consequently, the higher the entropy loading.

Overall, strong, positive, and significant correlations (significance of 0.027 or less) were found between the total number of objects, system entropy, and entropy loading on one side and most of the lines of code metrics class for both C and Ada programs on the other side. Additionally, strong, negative, and significant correlations (significance of 0.020 or less) exist between the average object entropy measure and most of the measures in the lines of code class for the Ada programs. The average object entropy is an indicator of the information contained inside objects.

### 7.2 Control Structure Entropy Analysis

While entropy loading is used as a measure of information among collections of objects, control structure entropy, defined in Section 3.2, is used as a measure of the complexity of a program. Consequently, a higher control structure entropy is expected as the perceived complexity of the program increases.

One of the assumptions that we are trying to validate (or at least provide support for) in this study is that the more code that is available for reuse, the less complex the resulting new program tends to be. This is expected because the internal complexity of the reusable modules is not seen by the programmer when the new program is being developed since the modules already exist and either have been previously understood or can be understood in one chunk.

### TABLE III

<table>
<thead>
<tr>
<th>(Ada)</th>
<th>LOC</th>
<th>Dec</th>
<th>Nde</th>
<th>Com</th>
<th>Bla</th>
<th>Tot</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConStrEnt</td>
<td>.8551</td>
<td>.8385</td>
<td>.8467</td>
<td>.6925</td>
<td>.5225</td>
<td>.8136</td>
</tr>
<tr>
<td>(Ada)</td>
<td>LOC</td>
<td>Dec</td>
<td>Nde</td>
<td>Com</td>
<td>Bla</td>
<td>Tot</td>
</tr>
<tr>
<td>ObjTot</td>
<td>.9860</td>
<td>.8693</td>
<td>.9886</td>
<td>.9616</td>
<td>.9279</td>
<td>.9921</td>
</tr>
<tr>
<td>AveObjEnt</td>
<td>-.7692</td>
<td>-.6495</td>
<td>-.7934</td>
<td>-.7427</td>
<td>-.6034</td>
<td>-.7619</td>
</tr>
<tr>
<td>SELtoOTRat</td>
<td>-.6368</td>
<td>-.5398</td>
<td>-.6619</td>
<td>-.6117</td>
<td>-.4529</td>
<td>-.6257</td>
</tr>
<tr>
<td>SysEntr</td>
<td>.9852</td>
<td>.9346</td>
<td>.9745</td>
<td>.9505</td>
<td>.8898</td>
<td>.9856</td>
</tr>
<tr>
<td>SysEntLoa</td>
<td>.8095</td>
<td>.7265</td>
<td>.7978</td>
<td>.7949</td>
<td>.8489</td>
<td>.8234</td>
</tr>
<tr>
<td>(C)</td>
<td>LOC</td>
<td>Dec</td>
<td>Nde</td>
<td>Bra</td>
<td>Com</td>
<td>Bla</td>
</tr>
<tr>
<td>ConStrEnt</td>
<td>.9284</td>
<td>.5944</td>
<td>.9510</td>
<td>.9101</td>
<td>.0077</td>
<td>.8900</td>
</tr>
<tr>
<td>(C)</td>
<td>LOC</td>
<td>Dec</td>
<td>Nde</td>
<td>Bra</td>
<td>Com</td>
<td>Bla</td>
</tr>
<tr>
<td>ObjTot</td>
<td>.9947</td>
<td>.8142</td>
<td>.9897</td>
<td>.2896</td>
<td>.9670</td>
<td>.9168</td>
</tr>
<tr>
<td>AveObjEnt</td>
<td>-.3292</td>
<td>-.6901</td>
<td>-.2442</td>
<td>-.8891</td>
<td>-.3626</td>
<td>-.3524</td>
</tr>
<tr>
<td>SELtoOTRat</td>
<td>-.1778</td>
<td>-.5906</td>
<td>-.0887</td>
<td>-.8856</td>
<td>-.2169</td>
<td>-.2141</td>
</tr>
<tr>
<td>SysEntr</td>
<td>.9785</td>
<td>.8549</td>
<td>.9609</td>
<td>.3942</td>
<td>.9656</td>
<td>.9142</td>
</tr>
<tr>
<td>SysEntLoa</td>
<td>.8537</td>
<td>.4589</td>
<td>.8969</td>
<td>-.2007</td>
<td>.8061</td>
<td>.7597</td>
</tr>
</tbody>
</table>
Strong, positive, significant correlations (significance of 0.003 or less) exist between the control structure entropy metric and the lines of code metric for both C and Ada programs. This should be expected as the lines of code metric is used to quantify the amount of software that needs to be developed anew, validating our assumption. In other words, the less code available for reuse, the more complex the new program tends to be.

The relation between MIN and the brace lines measure discussed in Section 6.1.1 was verified by a strong, positive, significant correlation (significance of 0.001) between them.

8. Conclusions and Future Work

The main theme of this study was to theoretically explore the relationships between software reusability and two information theory based metrics: entropy loading and control structure entropy. A survey of the open literature indicated that previous work in this area had not addressed the idea of quantifying software reuse with this type of metric.

Entropy loading was found to be inversely proportional to the amount of reuse present in the programs. Entropy loading was always smaller in the optimum reuse cases. This was corroborated by strong correlations found between entropy loading and the size of the resulting new program, measured by the lines of code metric. Consequently, entropy loading can presumably provide a mechanism for selecting the optimum reuse case among different possibilities for reuse.

Control structure entropy, a measure of the complexity of a program, was also found to be a good indicator of reuse. The optimum reuse case (higher reuse) was always found to be the one with the lowest control structure entropy. Strong correlations exist between control structure entropy and the size of the resulting new program, measured by the lines of code metric.

In conclusion, there seems to be a relation between entropy loading and software reuse, and between control structure entropy and software reuse. But, care should be taken not to make any generalizations since this study was not a controlled experiment and the sample sizes were not arrived at statistically. The intent of this study was only to determine if a possible relation between the metrics and software reuse existed.

Suggestions for future work include conducting a similar, but controlled and larger scale experiment to test the hypothesis that the notion of software reuse can be quantified. Perhaps, adding other software metrics to the ones used in this study and/or adding other programming languages would provide more insight. Possible candidates include other information theory based metrics such as residual complexity [20] and other metrics such as software science metrics [8].

Other work might also include software reuse instances where some of the objects overlap (i.e., there is a certain degree of harmless overkill involved in the objects that are being reused). In such a case, entropy loading cannot be applied but other measures can be applied (e.g., ([20], [24])).

Finally, automated tools can be developed to determine assumptions made by objects in the calculation of the entropy loading metric. This task is considered the most time consuming in the application of the entropy loading metric.

9. References


1 An interaction is viewed as an information transfer within or among subsystems [2].
2 A flow chart is strongly connected if there is a path from node a to node b for every pair of distinct nodes a and b [7]. A flow chart is proper if each node can be reached from the entry point of the program and if each control structure has only one entry and one exit.
3 Adjoined IF symbols are two IF symbols which can be connected without passing through any nodes which belong to a third IF symbol [6].
4 UNIX is a Trademark of AT&T Bell Laboratories.
5 An object a is visible to another object b if when b is in control of program execution, control of execution can be transferred from b to a.
6 Ada is a Trademark of the U.S. Department of Defense (Ada Joint Program Office).
7 A line of code is defined by Conte et al. [7] 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."
8 VAX and ULTRIX are Trademarks of Digital Equipment Corporation.
9 VMS is a Trademark of Digital Equipment Corporation.