Software Qual J (2007) 15:179–212
DOI 10.1007/s11219-006-9010-3

Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach

Edward B. Allen · Sampath Gottipati · Rajiv Govindarajan

Published online: 9 February 2007

Abstract Software development is fundamentally based on cognitive processes. Our motivating hypothesis is that amounts of various kinds of information in software artifacts may have useful statistical relationships with software-engineering attributes. This paper proposes measures of size, complexity and coupling in terms of the amount of information, building on formal definitions of these software-metric families proposed by Briand, Morasca, and Basili.

Ordinary graphs represent relationships between pairs of nodes. We extend prior work with ordinary graphs to hypergraphs representing relationships among sets of nodes. Some software engineering abstractions, such as set-use relations for public variables, are better represented as hypergraphs than ordinary (binary) graphs.

Traditional software metrics are based on counting. In contrast, we adopt information theory as the basis for measurement, because the design decisions embodied by software are information. This paper proposes software metrics of size, complexity, and coupling based on information in the pattern of incident hyperedges. For comparison, we also define corresponding counting-based metrics.

Three exploratory case studies illustrate some of the distinctive features of the proposed metrics. The case studies found that information theory-based software metrics make distinctions that counting metrics do not, which may be relevant to software engineering quality and process. We also identify situations when information theory-based metrics are simply proportional to corresponding counting metrics.

E. B. Allen (✉)
Department of Computer Science and Engineering, Box 9637, Mississippi State University, Mississippi State, Mississippi 39762, USA
e-mail: edward.allen@computer.org

S. Gottipati
Technation Software Consulting, Inc., 300 N Dakota Ave, Suite 505B, Sioux Falls, SD-57104, USA
e-mail: sampath@tnscinc.com

R. Govindarajan
Peri Software, Warrenville, New Jersey, USA
e-mail: rajiv.govindarajan@citigroup.com

Keywords Software metrics · Size · Complexity · Coupling · Properties of metrics · Measurement theory · Information theory · Entropy · Excess entropy · C++ case studies

1 Introduction

This paper defines measures of a program's size, complexity, and coupling in terms of the amount of information in hypergraph abstractions of that program. This paper also presents three case studies that explore the properties of the proposed metrics. We are motivated by the possibility that the amounts of various kinds of information in software may have useful statistical relationships with software quality attributes.

Software development is primarily an intellectual endeavor. We suspect that mistakes during software design and implementation have their roots in cognitive processes. For example, when making a design decision, it often is difficult to remember everything that is relevant. Our working hypothesis (Allen, 2002) is that amounts of various kinds of information may be associated with the potential for cognitive overload (Miller, 1956), because human memory and recall capabilities are limited (Hilgard et al., 1971). Information is stored in memory when a software abstraction is understood. The amount of such information may have useful statistical relationships with mistakes which in turn result in software faults (Hatton, 1997).

A software measurement method typically has two phases: (1) creation of an abstraction of the software, and (2) measurement of that abstraction. The abstraction is chosen to characterize the attribute of interest. The measure is designed to quantify the attribute of interest in a way that facilitates further analysis. For example, McCabe complexity (McCabe, 1976) is based on an abstraction of control flow, and is calculated from the number of nodes and edges in a control flowgraph. In this paper, the class of abstractions is hypergraphs, and the proposed measures are based on information theory, rather than counting. Information theory offers a way to calculate the amount of information in an abstract object, irrespective of perception by a person. Even though the total information of concern to a programmer is found in the combination of many kinds of abstractions of software, our research has focused on measurement of information in one abstraction at a time.

The software-engineering community often evaluates designs in terms of size, length, complexity, coupling, cohesion, etc. Briand, Morasca, and Basili propose definitions of these five attributes through a formal framework based on graphs (Briand et al., 1996b). The framework defines sets of properties for each of these five attributes. We support their proposal that the software-engineering community restrict the meaning of the attribute names of "size," "length," "complexity," "coupling," and "cohesion" to measures that conform to consensus properties. Software designs are often depicted by graphs. We adopt their idea that graph abstractions of software should be the basis for software measurement.

While strictly speaking a *graph* represents relationships between pairs of nodes, a *hypergraph* represents relationships among sets of nodes. Thus, a *hyperedge* may connect to multiple nodes, as shown in Fig. 1. Some software engineering abstractions, such as set-use relations for public variables, are better represented as hypergraphs than ordinary (binary) graphs (Schütt, 1977). In the remainder of this paper, we use the term *graph* in a broad sense to mean a hypergraph, unless otherwise indicated.

Traditional software metrics count software features (Fenton and Pfleeger, 1997). This seems to be based on the assumption that each item is equal in the mind of a developer. However, repetitive patterns are easier to remember than a random list. In contrast to counting, we adopt information theory (Cover and Thomas, 1991; Shannon and Weaver, 1949) as the basis for measurement (Allen, 1995). When symbolic content can be compactly described,

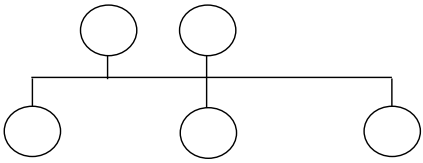

Fig. 1 Example hypergraph with one hyperedge

such as repetitive patterns, there is a small amount of information and high redundancy (Cover and Thomas, 1991). This paper proposes software metrics based on information in the pattern of incident hyperedges. For comparison, we also define corresponding counting-based metrics.

Prior work proposed information theory-based measures of size, length, complexity, coupling, and cohesion of ordinary graphs at the system level and at the module level (Allen, 2002; Allen and Khoshgoftaar, 1999; Allen et al., 2001; Chen, 2000). This paper extends this work proposing measures for size, complexity, and coupling of hypergraph abstractions of software, rather than ordinary graphs. This paper also contributes three case studies to elucidate the properties of the proposed metrics. Because we propose to measure hypergraphs, we can use the same formulas to measure many kinds of software abstractions, and thus, many kinds of software size, complexity, and coupling.

For convenient reference, Table 1 informally introduces basic terms used in this paper, and Table 2 defines notation. The following sections discuss related work, our working

Table 1 Definitions

Term	Definition
System	An abstraction of a software-development artifact, defined by a set of elements and a relation on them (Morasca and Briand, 1997). We restrict this abstraction to a hypergraph, consisting of nodes and hyperedges.
Node	Each node (or vertex) corresponds to an element.
Edge	Each edge (or arc) corresponds to a binary relationship between two nodes.
Hyperedge	Each hyperedge corresponds to a relationship among a subset of nodes.
System graph	A graph consisting of a disconnected node, modeling the environment, plus all the nodes and hyperedges in a system.
Hyperedges-only graph	A graph consisting of all nodes of a system that are connected to hyperedges, and all hyperedges, plus the environment node.
Undirected graph	A representation of a system where the direction of each hyperedge connection is ignored.
End points	The nodes connected to a hyperedge.
Nodes × hyperedges table	A table representing an undirected graph where each row signifies a node, each column denotes a hyperedge, and a 1 or 0 in a cell indicates that the hyperedge is incident to the node, or not, respectively.
Row pattern	The pattern of values in the cells of a row of a nodes × hyperedges table.
Label	A row pattern associated with a node.
Module	A subset of nodes and their incident hyperedges.
Disjoint modules	Two modules with no nodes in common and no path between them (Briand et al., 1996b).
Modular system	A partition of the nodes of a system into subsets, one subset for each module (Briand et al., 1996b). (The subsets of nodes do not intersect.)
Intermodule-hyperedges graph	A graph consisting of all the nodes in a modular system and all intermodule hyperedges, plus the environment node.

Definitions are ordered such that each definition does not use terms below it in the list.

Table 2 Notation

Symbol	Definition
System-related symbols	
S	System
MS	Modular system
S	System graph or its nodes × hyperedges table
$S^{\#}$	Hyperedges-only system graph
S^{*}	Intermodule-hyperedges system graph
S_0	System graph with no hyperedges
S_i	Subtable of $S^{\#}$ consisting of the columns where the ith row is not 0
Module-related symbols	
m_k	Module
n	Number of nodes in a system, **S**
n_M	Number of modules in *MS*
n_e	Number of hyperedges in a system, **S**
n_{e_k}	Number of hyperedges incident to nodes in module, m_k
$n_{\text{inter\_e}}$	Number of intermodule hyperedges in a system, **S**
$n_{\text{inter\_e\_k}}$	Number of intermodule hyperedges incident to nodes in module, m_k
Row-related symbols	
i, j	Indexes for a row in S, $i = 0, \ldots, n$ and similarly j. By convention, the environment node is indexed $i = 0$, and nodes in a system are indexed $i = 1, \ldots, n$.
k	Index for a module in **S**, $k = 1, \ldots, n_M$
l	Index for a pattern of values on a row (label)
$L(i)$	Function that determines the label of a row i
$L_i(j)$	Function that determines the label of a row j in S_i
Entropy-related symbols	
p	Probability mass function
$\hat{p}_l$	Proportion of the lth row pattern (estimated probability)
log	Logarithm, base 2
H	Entropy of a probability distribution
C	Excess entropy

abstractions for measurement, metric definitions and their rationale, exploratory case studies, a discussion, and conclusions.

2 Related work

Khoshgoftaar and Allen's survey (Khoshgoftaar and Allen, 1994) discusses numerous proposed software metrics based on information theory. Several are entropy-based measures of graph attributes, such as the entropy of control-flow graph nodes by their degree (Davis and LeBlanc, 1988), and the entropy of a data structure graph by its topology (Lew et al., 1988). From among the surveyed metrics, we extend Mohanty's work (Mohanty, 1981), which applied "excess entropy" to interaction among subsystems. The following summarizes some information theory based software metrics that have been proposed since their survey (Khoshgoftaar and Allen, 1994).

Bansiya et al. propose an entropy-based metric of software classes (Bansiya et al., 1999). Probability distributions are derived from the frequencies that name strings appear in a class's source code, and entropy is then calculated.

Abd-El-Hafiz proposes measures of "information content" of software (Abd-El-Hafiz, 2001). His abstraction of software is based on static frequencies of function calls in source code, from which probability distributions can be derived. Several kinds of entropy are proposed as software measures.

Kim, Shin, and Wu propose object-oriented software "complexity" measures based on entropy for classes, systems of objects, and the combination (Kim et al., 1995). Their abstraction of a class is a graph depicting function calls and set/use of data by functions within each class. Their abstraction of an object system is a graph representing message passing (method calls) among objects. The static frequency of these relationships is the basis for calculating probability distributions. With probability distributions in hand, each "complexity" is its entropy.

Shereshevshky et al. propose dynamic measures of "coupling" and "cohesion" based on entropy (Shereshevshky et al., 2001). Their abstraction of software is a graph based on information flows among components where the probability of each flow is based on operational use of the software, and thus, depends on program dynamics. Various kinds of "coupling" and "cohesion" are defined as the entropy of appropriate probability distributions of information flows.

Visaggio proposes an entropy-based metric for impact analysis considering artifacts from the requirements phase through the implementation phase (Visaggio, 1997). His abstraction is a graph representing traceability and dependency relationships among models of software from each phase of the software development life cycle. This graph allows one to identify "impact paths" for each proposed modification to the software, from requirements through implementation. Assuming impact paths are considered equally probable, the "structural information" is proportional to the entropy of the uniform distribution of impact paths.

Independent of our research, Chapin proposes a "complexity" measure based on excess entropy (Chapin, 2002), similar to our system complexity metric for ordinary graphs (Allen, 2002). His abstraction of software is a graph derived from message flows among software components. Chapin prefers using the amount of information ("entropy loading") as a measure, rather than just entropy; our approach is similar to Chapin on this point.

In contrast to metrics proposed by many others, we propose measures of the aggregate amount of information, rather than the average amount of information (i.e. entropy), and in contrast to our prior work, we apply them to hypergraphs, rather than just ordinary graphs. Hypergraphs can represent many different abstractions of software. For example, two of the case studies in this paper focused on relationships between methods and public variables in C++ source code.

3 Abstractions of software

During development, a large number of decisions are made on many levels of design and implementation. The software product is the accumulation of these decisions. Each decision can be viewed as an element of information. A design abstraction represents a set of design decisions that are of interest. For example, when a graph is an abstraction of software, the creation of each node and each edge are design decisions. Each connection is another design decision. Thus, software development entails a myriad of small decisions. Design artifacts, such as diagrams, are created during development to embody such abstractions.

Graphs directly used by designers are likely to be related to software quality (Andersson et al., 2003), for example, graphs produced by design tools. Graphs that are reverse engineered from code may depict relationships perceived by someone reading the source code (Runeson et al., 2006). The literature recommends many different metrics based on different abstractions. For example, Briand, Daly, and Wüst survey measures of coupling and cohesion for object-oriented designs (Briand et al., 1997a, 1999). They point out the various underlying abstractions, such as class inheritance, class type, method invocation, and attribute references. Our approach is compatible with abstractions from both the object oriented and procedural paradigms. Examples from the procedural paradigm include call graphs and set-use graphs for data structures. In our context, each combination of an abstraction and a metric measures a distinct attribute of the software.

Briand, Morasca, and Basili define an abstraction of a *system* as an ordinary graph (Briand et al., 1996b). Morasca and Briand extend this to relations in general (Morasca and Briand, 1997). We restrict our study to relations represented by hypergraphs where the order of elements in the relation does not matter. Software engineers often identify each node with a component name. Consequently, we choose working abstractions that maintain a distinct identity for each relationship, rather than an abstraction of topology alone (Lew et al., 1988). The software-engineering meanings of elements (nodes) and relations (hyperedges) is separately specified for each practical application. Our proposed metrics, defined below, are applicable whenever the real-world system is represented by a hypergraph.

Definition 1 (System and module). A system, **S**, is an abstraction of a software system represented by a graph with $n > 0$ nodes and with $n_e \geq 0$ hyperedges connecting zero or more of the nodes. A subset of nodes and their incident hyperedges may be designated as a module, m_k.

In this paper, undirected graphs are sufficient for our purposes, and the topology is not restricted (e.g., acyclic graphs are not required). For example, Fig. 2 depicts a hypergraph with four modules. In this example, hyperedges 1, 2, 3, and 8 have more than two connections, but the other hyperedges have only two end points, i.e. they are ordinary edges.

Given an abstraction of software represented by a graph, **S**, various subgraphs are working abstractions in this paper. We extend the system's graph abstraction, **S**, by defining a *system graph*, *S*, to model explicitly the lack of relationship between the system and its environment.

Definition 2 (System graph) (Allen and Khoshgoftaar, 1999). The system graph, *S*, of a system, **S**, with n nodes, is all nodes in **S** and all its hyperedges, plus a disconnected node modeling the system's environment. Without loss of generality, index the environment node as $i = 0$, and the nodes in **S** as $i = 1, \ldots, n$.

Our goal is to analyze patterns of connections in a system graph. We label each node with the set of hyperedges that are incident to it. We want to work with a convenient tabular representation of a graph that fully specifies an undirected graph. We choose a nodes $\times$ hyperedges table where each cell indicates whether the node is connected to the hyperedge, or not, encoded as one or zero respectively (Allen and Khoshgoftaar, 1999). Consequently, each node's label (i.e., the set of incident hyperedges) is encoded as the binary pattern of values in a row of the table. The abstraction of a nodes $\times$ hyperedges table is essentially an object predicate table (van Emden, 1970), where nodes are objects and each predicate is of the form, "Is this node related to other nodes by this hyperedge?"

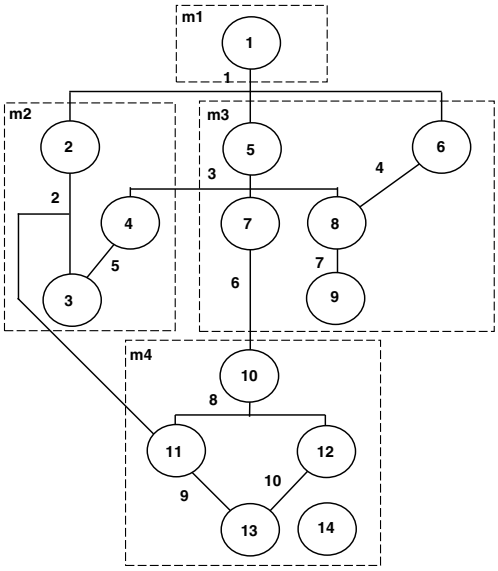

Fig. 2 Example of hypergraph, **S**

Table 3 is the nodes × hyperedges table for Fig. 2. Columns of the binary pattern represent hyperedges, numbered left to right. Complexity is defined by Briand, Morasca, and Basili in terms of edges (Briand et al., 1996b). To support measurement of complexity, we make a working abstraction, a *hyperedges-only graph*, $\mathbf{S}^{\#}$.

Definition 3 (Hyperedges-only graph) (Allen, 2002). Given a system, **S**, its hyperedges-only graph, $\mathbf{S}^{\#}$, consists of all nodes in **S** connected to hyper edges and all its hyperedges. The corresponding system graph is denoted by $S^{\#}$.

In other words, $S^{\#}$ is constructed from S by deleting all isolated nodes except the environment node. The hyperedges-only graph of the system in Fig. 2 is similar to the system graph,

Table 3 Nodes × Hyperedges table for example system

Module	Node	Hyperedges
env.	0	0000000000
m_1	1	1000000000
m_2	2	1100000000
m_2	3	0100100000
m_2	4	0010100000
m_3	5	1010000000
m_3	6	1001000000
m_3	7	0010010000
m_3	8	0011001000
m_3	9	0000001000
m_4	10	0000010100
m_4	11	0100000110
m_4	12	0000000101
m_4	13	0000000011
$m4$	14	0000000000

Table 4 Nodes × hyperedges table for example $S^{\#}$

Module	Node	Hyperedges
env.	0	0000000000
m_1	1	1000000000
m_2	2	1100000000
m_2	3	0100100000
m_2	4	0010100000
m_3	5	1010000000
m_3	6	1001000000
m_3	7	0010010000
m_3	8	0011001000
m_3	9	0000001000
m_4	10	0000010100
m_4	11	0100000110
m_4	12	0000000101
m_4	13	0000000011

except that Node 14 is omitted. It is not connected to any hyperedge, so it is not included. Its nodes × hyperedges table is in Table 4.

Coupling is defined by Briand, Morasca, and Basili for a graph partitioned into subsystems (modules) (Briand et al., 1996b), such as Fig. 2 above.

Definition 4 (Modular system) (Briand et al., 1996b). A modular system, *MS*, is a special case of a system, **S**, whose n nodes are partitioned into n_M modules, m_k, $k = 1, \ldots, n_M$.

This means every node is in a module and no node is in multiple modules.

To support measurement of coupling, we make a working abstraction, an *intermodule-hyperedges graph*, MS^*.

Definition 5 (Intermodule-hyperedges graph) (Allen and Khoshgoftaar, 1999). Given a modular system, *MS*, its intermodule-hyperedges graph, MS^*, consists of all nodes in *MS* and all its intermodule hyperedges. The corresponding system graph is denoted by S^*.

For example, Fig. 3 depicts the intermodule-hyperedges hyperedges-only graph, $S^{*\#}$, for the modular system in Fig. 2, and Table 5 depicts the corresponding nodes × hyperedges table.

Table 5 Nodes × hyperedges table for example $S^{*\#}$

Module	Node	Hyperedges
env.	0	0000
m_1	1	1000
m_2	2	1100
m_2	3	0100
m_2	4	0010
m_3	5	1010
m_3	6	1000
m_3	7	0011
m_3	8	0010
m_4	10	0001
m_4	11	0100

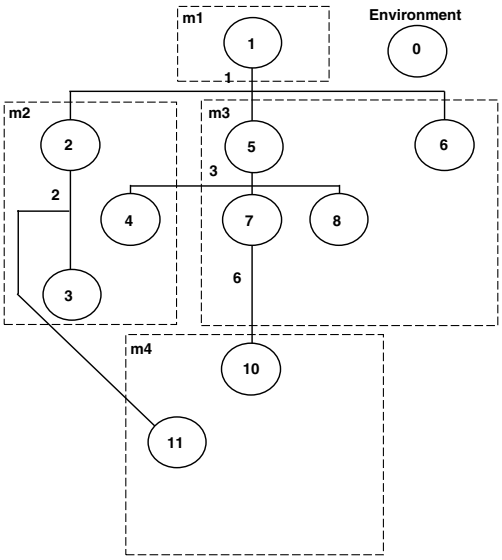

Fig. 3 Example intermodule-hyperedges hyperedges-only graph, $S^{*\#}$

4 Size

Research in software metrics has found that "size" is one of the dominant attributes when predicting software quality (El Emam et al., 2001; Munson and Khoshgoftaar, 1989). In this paper, we focus on the size of a hypergraph in the abstract. The real-world meaning of the hypergraph is the key to interpreting its size measurements.

4.1 Properties of size metrics

Briand, Morasca, and Basili propose a set of properties that defines the concepts of the *size of a system* and the *size of a module* (Briand et al., 1996b). Table 6 summarizes the properties of this family of measures. In a software metrics context, we reserve the term *size* for measures that have these properties. Briand, Morasca, and Basili also state the corollaries in Table 7 (Briand et al., 1996b).

Table 6 Properties of size (Briand et al., 1996b)

System properties

1. Nonnegativity. The size of a system is nonnegative.
2. Null value. The size of a system is zero if its set of nodes is empty.
3. Module additivity. Given a system, **S**, having modules, m_1 and m_2, such that every node in **S** is in m_1 or m_2, but not both, the size of this system is equal to the sum of the sizes of the modules m_1 and m_2.

$$Size(\mathbf{S}) = Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

Module properties

4. Nonnegativity. The size of a module is nonnegative.
5. Null value. The size of the module is zero if its set of nodes is empty.
6. Monotonicity. Adding a node to a module does not decrease its size.

Table 7 Corollaries for size metrics (Briand et al., 1996b)

1. Node additivity. Given a modular system, MS, where each node is a module, m_k, $k = 1, \ldots, n_M$, the size of the modular system is given by

$$Size(MS) = \sum_{k=1}^{n_M} Size(m_k|MS)$$

2. Monotonicity. Adding a node to a system does not decrease its size.
3. General module additivity. Given a system, **S**, with any two modules, m_1 and m_2 such that every node in **S** is a node in m_1 or m_2 or both, the size of the system is not greater than the sum of the sizes of the pair of modules.

$$Size(\mathbf{S}) \leq Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

4. Merging of modules. Given a system, **S**, with any two modules, m_1 and m_2 such that every node in **S** is a node in m_1 or m_2 or both, construct $\mathbf{S}'$ such that m_1 and m_2 in **S** are replaced by $m_{1\cup 2} = m_1 \cup m_2$ in $\mathbf{S}'$.

$$Size(\mathbf{S}') \leq Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

4.2 Counting-based size metrics

Morasca and Briand imply counting-based size metrics by asserting that the size of a system depends on the elements (nodes) of the system and not its relations (edges) (Morasca and Briand, 1997). Accordingly, systems with the same set of elements have the same size regardless of their set of relationships (edges). Our counting-based size metrics have this property.

Definition 6 (Counting size of a system). The *counting size* of a system **S** is the number of nodes in **S**.

$$CountingSize(\mathbf{S}) = n \tag{1}$$

Definition 7 (Counting size of a module). The counting size of a module for a system **S** is the number of nodes in the module.

$$CountingSize(m_k|\mathbf{S}) = n_k \tag{2}$$

Several measures can be classified as size measures, such as lines of code, number of modules, number of procedures, etc. (Briand et al., 1996b). Each of the counted entities of an abstraction is modeled as a node. In our case studies below, counting size of a module is the number of methods in a class. This corresponds to Weighted Methods per Class (WMC) (Chidamber and Kemerer, 1994) with weights of one.

4.3 Information theory-based size metrics

A theory for the information content of finite strings of characters was independently invented by Solomonoff (1964), Kolmogorov (1965), and Chaitin (1966). The name Kolmogorov

complexity has become commonly understood for this field (Li and Vitányi, 1988). Informally, the Kolmogorov complexity, K, of a string of characters is the size of the smallest Turing machine program that outputs that string and then terminates. In this theoretical context, Kolmogorov complexity is an attribute of the string of characters.

Such a theory is attractive for objectively quantifying the information in a software abstraction, but there is a practical problem. The function K is not partial recursive, i.e., it is not computable exactly (Li and Vitányi, 1988). Therefore, we must be satisfied with an approximation.

Kolmogorov complexity and information theory (Shannon and Weaver, 1949) are intimately related (Chaitin, 1975; Kolmogorov, 1968). For a random variable x, let $H(x)$ be the symbol for the *entropy* of the distribution of x. Let n_x be the number of items in the domain of x, and let p_l be the probability of an item l from that domain. Entropy is calculated by the following.

$$H(x) = \sum_{l=1}^{n_x} p_l(-\log p_l) \tag{3}$$

where the base of the logarithm is the number of symbols in the alphabet used to encode the domain of x (e.g., two for binary encoding). We do not assume all items in the domain are equally probable.

An *instantaneous code* is a set of strings (code words) where no string in the set is a prefix of another, and each code word corresponds to an item in the domain of x. Let $E(Len(x))$ be the expected length per item of an instantaneous code for the domain of x. Information theory gives the following result (Cover and Thomas, 1991, p. 86).

$$E(Len(x)) \geq H(x) \tag{4}$$

In other words, the entropy of the distribution of x is the minimum expected length of an instantaneous code for one sample item.

If we use an instantaneous code to describe the domain of x, and if a set X consists of n independent identically distributed items from the domain of x, then the *minimum expected length of a description* of X is $nH(x)$. Based on the length of Shannon-Fano encoding (Cover and Thomas, 1991, p. 170), this is an approximation to the Kolmogorov complexity of the set of objects.

$$\hat{K}(X) = nH(x) \tag{5}$$

In this context, the base of the logarithm in Eq. (3) is the number of symbols in the code alphabet. Therefore, we interpret this as an approximation of the total information in the set. Kolmogorov complexity focuses on the product, and therefore, does not address variation in comprehension skills among individuals.

Suppose we have a system, **S**. We model the designer's preferences for connecting nodes to hyperedges as a probability distribution. Specifically, we model its system graph S as a set of statistically independent identically distributed samples from a probability distribution on the possible row patterns of its nodes $\times$ hyperedges table, $p_l, l = 1, \ldots, n_S$, where n_S is the number of possible distinct row patterns. The *entropy* of the distribution of row patterns

Table 8 Probabilities of row patterns in S

Nodes	Hyperedges	$\hat{p}_l$
0, 14	0000000000	2/15
1	1000000000	1/15
2	1100000000	1/15
3	0100100000	1/15
4	0010100000	1/15
5	1010000000	1/15
6	1001000000	1/15
7	0010010000	1/15
8	0011001000	1/15
9	0000001000	1/15
10	0000010100	1/15
11	0100000110	1/15
12	0000000101	1/15
13	0000000011	1/15

See Table 3.

(Shannon and Weaver, 1949) is the following.

$$H(S) = \sum_{l=1}^{n_S} p_l(-\log p_l) \tag{6}$$

Table 8 lists the row patterns that occur in Table 3. The pattern 0000000000 occurs twice among 15 nodes, and therefore we estimate the probability of that row pattern as $\hat{p}_0 = 2/15$. The other thirteen row patterns are each unique, so we estimate each probability as $\hat{p}_l = 1/15$. By Eq. (6), the estimated entropy is

$$\hat{H}(S) = \frac{2}{15}\left(-\log\frac{2}{15}\right) + 13\left(\frac{1}{15}\left(-\log\frac{1}{15}\right)\right) = 3.77 \text{ bits} \tag{7}$$

In this application, entropy is the average information per node. Because the nodes $\times$ hyperedges table has binary values, we use base 2 logarithms in information theoretic calculations. Consequently, the unit of measure is a *bit*. A bit is the commonly used measure of information in the communications field.

The number of rows including the environment is $n+1$. Let n_l be the number of rows with pattern l. The proportion of rows with pattern l is an estimate of the probability of pattern l (van Emden, 1970), namely, $\hat{p}_l = n_l/(n+1)$. By Eq. (6), entropy can be estimated by the following.

$$\hat{H}(S) = \sum_{l=1}^{n_S} \frac{n_l}{n+1}\left(-\log\frac{n_l}{n+1}\right) \tag{8}$$

$$\hat{H}(S) = \sum_{i=0}^{n} \frac{1}{n+1}\left(-\log \hat{p}_{L(i)}\right) \tag{9}$$

where $L(i)$ is a function that gives the label l (i.e., row pattern) of node i. Note that the summation in Eq. (9) is over the set of nodes (i), rather than the set of distinct row patterns (l), as in Eq. (6).

If we model the number of nodes, n, and the number of hyperedges, n_e, as given,[1] and, without loss of generality, if we use an instantaneous code to describe row patterns, then the minimum expected length of a description (Cover and Thomas, 1991) of a set S consisting of $n+1$ nodes is $(n+1)\,H(S)$.

$$(n+1)\hat{H}(S) = \sum_{i=0}^{n} \left(-\log \hat{p}_{L(i)}\right) \tag{10}$$

Thus, each node contributes to the minimum expected length of a description of S. Because we estimate probabilities by proportions $\hat{p}_l = n_l/(n+1)$, the existence of the environment node assures that the estimated probability of a disconnected node is nonzero, $\hat{p}_{L(0)} \geq 1/(n+1) > 0$. This facilitates monotonicity properties of metrics. We interpret the minimum expected length of a description of S as the amount of information in S. This leads to our definition of *size*.

Definition 8 (Information size of a system) (Allen, 2002). The size of a system, **S**, is the amount of information in its system graph, S, less the contribution of the environment.

$$\mathit{Size}(\mathbf{S}) = (n+1)H(S) - \left(-\log p_{L(0)}\right) \tag{11}$$

The estimated information contribution of the environment node is $-\log \hat{p}_{L(0)}$. Consequently, by Eq. (10), the size of a system is estimated by the following.

$$\text{estimated } \mathit{Size}(\mathbf{S}) = \sum_{i=1}^{n} \left(-\log \hat{p}_{L(i)}\right) \tag{12}$$

The summation begins with $i=1$ instead of $i=0$. Applying Eq. (12) to the probabilities in Table 8, summing over Nodes 1 through 14.

$$\text{estimated } \mathit{Size}(\mathbf{S}) = 13\left(-\log \frac{1}{15}\right) + \left(-\log \frac{2}{15}\right) = 53.7 \text{ bits} \tag{13}$$

A portion of the system's information may be attributed to each module.

Definition 9 (Information size of a module) (Allen, 2002). The size of a module, m_k, in a system, **S**, is the information in its system graph contributed by the module.

$$\mathit{Size}(m_k|\mathbf{S}) = \sum_{i \in m_k} \left(-\log p_{L(i)}\right) \tag{14}$$

Figure 2 has the size measurements in Table 9. This illustrates Corollary 1 in Table 7: the system size is the sum of the module sizes.

The more row patterns are repeated in a nodes $\times$ hyperedges table, the smaller information size becomes. In this way, information size is inherently different from counting size.

[1] As a simplifying assumption, our analysis excludes from measurement the design decisions of how many nodes and how many hyperedges the graph includes.

Table 9 Size measurements of example

Object measured	Information (bits)	Count (nodes)
S	53.7	14
m_1	3.9	1
m_2	11.7	3
m_3	19.6	5
m_4	18.5	5

In the extreme case when a graph, S_0, has no hyperedges, its nodes $\times$ hyperedges table is equivalent to a table with an arbitrary number of columns and a zero in every cell. Because $\hat{p}_0 = 1$, the contribution of the environment node is zero, $-\log \hat{p}_0 = 0$, the estimated entropy of the nodes $\times$ hyperedges table is zero, $\hat{H}(S_0) = 0$, and thus, the estimated information size is zero, $Size(S_0) = 0$, when there are no hyperedges. In contrast, the counting size is the number of nodes, even if there is no hyperedge.

5 Complexity

The software metrics literature is rife with controversy over the term "complexity." A multitude of metrics have been proposed over the years (Zuse, 1997b). Briand, Morasca, and Basili define complexity as an intrinsic attribute of a graph abstraction and not its psychological complexity as perceived by an external observer (Briand et al., 1996b). In this paper, we focus on relationships as complexity's underlying concept.

5.1 Properties of complexity metrics

Briand, Morasca, and Basili propose a set of properties that defines the concepts of the *complexity of a system* and the *complexity of a module* (Briand et al., 1996b). Table 10 presents our proposed properties of complexity for hypergraphs. Our properties agree with Briand, Morasca, and Basili, except we require a slightly stronger constraint in Property 4: we require "no nodes in common," but Briand, Morasca, and Basili require "no edges in common" in their corresponding property. Briand, Morasca, and Basili's properties are based on directed graphs. Property 3, based on their work, asserts that the direction of hyperedge connections is not relevant to complexity. A nodes $\times$ hyperedges table represents an undirected hypergraph, and thus, the metrics are consistent with Property 3. In a software metrics context, we reserve the term *complexity* for measures that have these properties. Briand, Morasca, and Basili also state the corollary in Property 6 of Table 10.

5.2 Counting-based complexity metrics

Examples by Briand, Morasca, and Basili guided us in defining counting-based complexity metrics (Briand et al., 1996b).

Definition 10 (Counting complexity of a system). The counting complexity of a system **S** is the number of hyperedges in the system.

$$CountingComplexity(\mathbf{S}) = n_e \tag{15}$$

Table 10 Properties of complexity

System properties

1. Nonnegativity. The complexity of a system is nonnegative.
2. Null value. The complexity of a system is zero if its set of hyperedges is empty.
3. Symmetry. The complexity of a system does not depend on the convention chosen to represent the direction of hyperedge connections.
4. Module monotonicity. Given a System, **S**, with any two modules, m_1 and m_2, that have no nodes in common, the complexity of the system is no less than the sum of the complexities of the two modules.

$$Complexity(\mathbf{S}) \geq Complexity(m_1|\mathbf{S}) + Complexity(m_2|\mathbf{S})$$

5. Disjoint module additivity. Given a system, **S**, composed of two disjoint modules, m_1 and m_2, the complexity of the system is equal to the sum of the complexities of the two modules.

$$Complexity(\mathbf{S}) = Complexity(m_1|\mathbf{S}) + Complexity(m_2|\mathbf{S})$$

6. Corollary: Monotonicity. Adding a hyperedge to a system does not decrease its complexity.

Module properties

7. Nonnegativity. The complexity of a module is nonnegative.
8. Null value. The complexity of a module is zero if its set of intermodule and intramodule hyperedges is empty.
9. Monotonicity. Adding an intermodule or intramodule hyperedge to a module does not decrease its complexity.

Definition 11 (Counting complexity of a module). The counting complexity of a module m_k in a system **S** is the number of hyperedges incident to nodes in the module.

$$CountingComplexity(m_k|\mathbf{S}) = n_{\mathrm{e\_k}} \tag{16}$$

The data-flow complexity measure proposed by Oviedo is an example measure that counts the number of direct and indirect variable definitions (i.e., setting a value) in a block of a program that can directly reference the set of variables (Oviedo, 1980). This data-flow complexity satisfies all of the properties of Briand, Morasca, and Basili, and thus, is a complexity measure according to their definition (Briand et al., 1996b).

5.3 Information theory-based complexity metrics

Because the properties of complexity focus on hyperedges, we define complexity as a measurement of a hyperedges-only graph, $\mathbf{S}^{\#}$. Consider the following subgraph of $\mathbf{S}^{\#}$.

Definition 12 (Node subgraph) (Allen and Khoshgoftaar, 1999). Given a hyperedges-only graph, $\mathbf{S}^{\#}$, the node subsystem graph, $\mathbf{S}_i$, consists of all the nodes in $\mathbf{S}^{\#}$ and the hyperedges of $\mathbf{S}^{\#}$ connected to the ith node. Its system graph is denoted by S_i.

Similar to S, we label each node with the set of hyperedges incident to it, and we represent S_i by a nodes $\times$ hyperedges table. Figure 4 is an example of a node subgraph for Node 2, $\mathbf{S}_2$, in $\mathbf{S}^{\#}$, and its nodes $\times$ hyperedges table is Table 11. Hyperedges 1 and 2 are the only ones in Table 4 that are incident to Node 2. Disconnected nodes are included in $\mathbf{S}_i$, but are not drawn in Fig. 4.

Table 11 Nodes × hyperedges table for example node subgraph, S_2

Module	Node	Hyperedges
env.	0	00
m_1	1	10
m_2	2	11
m_2	3	01
m_2	4	00
m_3	5	10
m_3	6	10
m_3	7	00
m_3	8	00
m_3	9	00
m_4	10	00
m_4	11	01
m_4	12	00
m_4	13	00

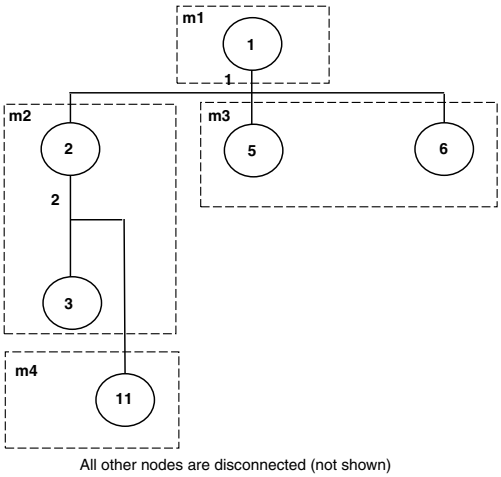

Fig. 4 Example node subgraph, $\mathbf{S}_i$, for Node 2

For size, we model S as a probability distribution on its row patterns. Similarly, we also model S_i as a probability distribution, estimated by the proportions of distinct row patterns, $\hat{p}_l$. Similar to Eq. (9), we estimate the entropy of the distribution of row patterns by the following.

$$\hat{H}(S_i) = \sum_{j=0}^{n} \frac{1}{n+1}\left(-\log \hat{p}_{L_i(j)}\right) \tag{17}$$

where $L_i(j)$ is a function that gives the pattern index, l, of the jth row of S_i.

Each row of each S_i is a subset of the corresponding row of $S^{\#}$. $S^{\#}$ represents the joint distribution of all the S_i. Information theory states that the entropy of a joint distribution is less than or equal to the sum of the entropy of the components.

$$\sum_{i=0}^{n} H(S_i) \geq H(S^{\#}) \tag{18}$$

Watanabe shows that the difference is a measure of the relationships among the components (Watanabe, 1960). *Excess entropy* (van Emden, 1970) of $S^{\#}$ is defined as

$$C(S^{\#}) = \sum_{i=0}^{n} H(S_i) - H(S^{\#}) \tag{19}$$

Excess entropy is the average information in relationships. Connected nodes are related to each other by the presence of a hyperedge. If the S_i are highly related to each other by common hyperedge connections and common disconnected nodes, then the excess entropy is high.

Mohanty proposes to measure interactions among subsystems by excess entropy (Mohanty, 1979, 1981). We extend his approach to measure complexity and coupling (Allen, 2002; Allen and Khoshgoftaar, 1999). Rather than information per subsystem, we are interested in the amount of information overall (Chapin, 2002). Multiplying excess entropy by the number of nodes yields the amount of information in the relationships among the nodes, $(n+1)C(S^{\#})$.

Definition 13 (Information complexity of a system) (Allen, 2002). The complexity of a system, **S**, is given by the amount of information in relationships in its hyperedges-only graph, less the contribution of the environment.

$$Complexity(\mathbf{S}) = \sum_{i=1}^{n} Size(\mathbf{S}_i^{\#}) - Size(\mathbf{S}^{\#}) \tag{20}$$

When probabilities refer to $S^{\#}$ and its S_i, note that by Eqs. (9), (12), (17), and (19),

$$\text{estimated } Complexity(\mathbf{S}) = \left(\sum_{i=1}^{n} \sum_{j=1}^{n} \left(-\log \hat{p}_{L_i(j)} \right) \right) - \sum_{i=1}^{n} \left(-\log \hat{p}_{L(i)} \right) \tag{21}$$

$$\text{estimated } Complexity(\mathbf{S}) = (n+1)\hat{C}(S^{\#}) - \left(\sum_{i=1}^{n} \left(-\log \hat{p}_{L_i(0)} \right) - \left(-\log \hat{p}_{L(0)} \right) \right) \tag{22}$$

Definition 14 (Information complexity of a module) (Allen, 2002). The complexity of a module, m_k, in a system, **S**, is its contribution to the complexity of the system, given by

$$Complexity(m_k|\mathbf{S}) = \sum_{i \in m_k} Size(\mathbf{S}_i^{\#}) - Size(m_k|\mathbf{S}^{\#}) \tag{23}$$

Figure 2 has the complexity measurements shown in Table 12. The number of nodes in $S^{\#}$ is 14, including the environment, but not Node 14. The row pattern for every row in Table 4 is unique, and therefore, $\hat{p}_l = 1/14$ for Nodes 0 through 13. The information column is the value of *Complexity* for the object measured. This illustrates that the information system complexity is the sum of the information module complexities. This is not true for counting complexities, because counting module complexity includes incident intermodule hyperedges for each module, sometimes counting the same hyperedge in two or more modules.

Table 12 Complexity measurements of example

Object measured	Information (bits)	Count (hyperedges)
S	189.1	10
m_1	7.8	1
m_2	47.3	4
m_3	74.6	5
m_4	59.4	5

Table 13 Example where module 1 has negative module complexity

Module	Node	Hyperedges
env.	0	00000
m_1	**1**	**10000**
m_2	2	**1**1000
m_2	3	**1**0110
m_3	4	**1**0001

$Complexity(m_1|\mathbf{S}) = -1.0$ bits
$Complexity(m_2|\mathbf{S}) = 4.4$ bits
$Complexity(m_3|\mathbf{S}) = 2.2$ bits
$Complexity(\mathbf{S}) = 5.6$ bits

We found that this definition of module complexity does not conform to Property 7 regarding module nonnegativity in Table 10 for certain extreme cases. If a module consists of a single node in which hyperedges connected to this node are also connected to all other nodes in the system, then the complexity of this module will be negative, such as the example in Table 13 (except in the case of a completely connected hypergraph).

Module m_1 has a single node. Hyperedge 1 is connected to all the nodes in the system and the remaining hyperedges are not connected to the node in module m_1. In this case, $Complexity(m_1|\mathbf{S}) = -1.0$ bits < 0, which does not conform to Property 7 in Table 10, which in turn, means that it also does not conform to Property 9. Future research will investigate the practical importance of such extreme cases.

6 Coupling

A large literature on coupling (Briand et al., 1999) highlights the multitude of mechanisms that can cause one module to be related to another. In this paper, we focus on intermodule relationships in the abstract (Briand et al., 1996b). A wide variety of coupling mechanisms can be abstractly represented by a hypergraph. Our metrics are applicable to any such abstraction, yielding a wide variety of measures that are computed in a similar manner.

6.1 Properties of coupling metrics

Briand, Morasca, and Basili propose sets of properties that define the concepts of the *coupling of a modular system*, and the *coupling of a module* (Briand et al., 1996b). Table 14 summarizes properties of coupling of a modular system extended for hypergraphs.

In contrast to Briand et al. (1996), we make no distinction between inbound and outbound coupling, because a nodes $\times$ hyperedges table does not distinguish direction. In Properties 2, 3, 7, and 8 of Table 14, our properties say "intermodule" hyperedges, whereas the

Table 14 Properties of coupling

System properties
1. Nonnegativity. Coupling of a modular system is nonnegative.
2. Null value. Coupling of a modular system is zero if its set of intermodule hyperedges is empty.
3. Monotonicity. Adding an intermodule hyperedge to a modular system does not decrease its coupling.
4. Merging of modules. If two modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the coupling of the modular system with $m_{1\cup 2}$ is not greater than the coupling of the modular system with m_1 and m_2.
5. Disjoint module additivity. If two modules, m_1 and m_2, which have no intermodule hyperedges between nodes in m_1 and nodes in m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the coupling of the modular system with $m_{1\cup 2}$ is equal to the coupling of the modular system with m_1 and m_2.
Module properties
6. Nonnegativity. Coupling of a module is nonnegative.
7. Null value. Coupling of a module is zero if its set of intermodule hyperedges is empty.
8. Monotonicity. Adding an intermodule hyperedge to a module does not decrease its module coupling.
9. Merging of modules. If two modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the module coupling of $m_{1\cup 2}$ is not greater than the sum of the module coupling of m_1 and m_2.
10. Disjoint module additivity. If two modules, m_1 and m_2, which have no intermodule hyperedges between nodes in m_1 and nodes in m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the module coupling of $m_{1\cup 2}$ is equal to the sum of the module coupling of m_1 and m_2.

corresponding properties of Briand, Morasca, and Basili say "output" edges (Briand et al., 1996b).

6.2 Counting-based coupling metrics

Coupling captures the amount of relationship between nodes belonging to different modules of a system. Examples from the literature (Briand et al., 1999) guided our definitions of counting-based coupling metrics.

Definition 15 (Counting coupling of a system). The counting coupling of a modular system *MS* is the number of intermodule hyperedges in the system.

$$CountingCoupling(\mathbf{S}) = n_{\text{inter\_e}} \tag{24}$$

Definition 16 (Counting coupling of a module). The counting coupling of a module m_k in a modular system MS is the number of intermodule hyperedges incident to nodes in the module.

$$CountingCoupling(m_k|\mathbf{S}) = n_{\text{inter\_e\_k}} \tag{25}$$

In our case studies below, we focused on use of public variables. This counting coupling is part of the idea for Coupling Between Objects (CBO) (Chidamber and Kemerer, 1994), which also includes use of methods.

Table 15 Coupling measurements of example

Object measured	Information (bits)	Count (hyperedges)
S	89.9	4
m_1	7.3	1
m_2	28.8	3
m_3	43.6	3
m_4	10.2	2

6.3 Information theory-based coupling metrics

The properties of coupling focus on intermodule hyperedges. Consequently, we define coupling as a measure on an intermodule-hyperedges graph, MS^*. Our definition of *coupling* of a modular system builds on our definition of complexity (Allen and Khoshgoftaar, 1999).

Definition 17 (Information coupling of a modular system). The coupling of a modular system, *MS*, is the amount of information in intermodule relationships in its system graph, less the contribution of the environment.

$$Coupling(MS) = Complexity(MS^*) \tag{26}$$

Definition 18 (Information coupling of a module). The coupling of a module, m_k, in a modular system, *MS*, is its contribution to the coupling of the system, given by

$$Coupling(m_k|MS) = Complexity(m_k|MS^*) \tag{27}$$

Figure 2 has the coupling measurements shown in Table 15, based on the $S^{*\#}$ of Fig. 3. The Information column in Table 15 is the value of *Coupling* for the objects measured.

7 Case studies

This section provides exploratory case studies of (1) a set of artificially generated graphs (Gottipati, 2003), (2) a data manipulation program for a physics research project (Gottipati, 2003), and (3) selected source files from a mathematical library (Govindarajan, 2004).

7.1 Artificial examples

This case study considered small artificial hypergraphs to explore the properties of our metrics. This study illucidated some patterns and principles that also apply to larger systems.

Figures 5 through 8 show graphs that we generated for this case study. Figure 5 depicts a series of small graphs where a node is added and then a hyperedge is added. Figure 6 depicts three series of graphs where hyperedges are added which have the same connections as existing hyperedges. Table 16 presents the system-level metrics for these two figures. In these small graphs, every node is considered a module. Coupling measurements are the same as complexity, because every hyperedge is an intermodule hyperedge.

Table 16 System-level measurements of artificial examples

System	Size		Complexity	
	Information (bits)	Count (nodes)	Information (bits)	Count (hyperedges)
Test10	1.2	2	1.2	1
Test15	3.0	3	1.2	1
Test17	4.0	3	5.3	2
Test10	1.2	2	1.2	1
Test11	1.2	2	1.2	2
Test12	1.2	2	1.2	3
Test13	1.2	2	1.2	4
Test14	1.2	2	1.2	5
Test16	1.3	3	2.5	1
Test18	1.3	3	2.5	2
Test20	1.3	3	2.5	3
Test17	4.0	3	5.3	2
Test19	4.0	3	5.3	3

The graphs in Fig. 5 illustrate how adding a node increases information size, but not information complexity, and how adding a hyperedge increases both information size and information complexity.

The graphs in Fig. 6 illustrate that our information theory-based measurements are not sensitive to multiple hyperedges connected to exactly the same nodes. Redundant hyperedges do not add information in this probabilistic model, because the estimated probabilities of row patterns are not affected by redundant hyperedges.

Graph test16 in Fig. 6 has a smaller information size than test15 in Fig. 5, because the nodes in test16 have a symmetric pattern of connections which in turn, is described by less information than the pattern in test15.

Figure 7 depicts two pairs of binary trees. Trees 1a and 2a have ordinary edges (two connections per edge). Trees 1b and 2b have hyperedges with three connections per hyperedge. Figure 8 depicts two pairs of (nonbinary) trees. Trees 3a and 4a have ordinary edges (two connections per edge). Trees 3b and 4b have hyperedges with more than two connections per hyperedge. Table 17 presents the system-level metrics for these two figures. Abstractions of software using ordinary edge smake a distinction for each edge relationship. Abstractions using hyperedges are appropriate when such distinctions are not relevant, and thus, information size is smaller. However, we see that information complexities are about the same for the larger graphs.

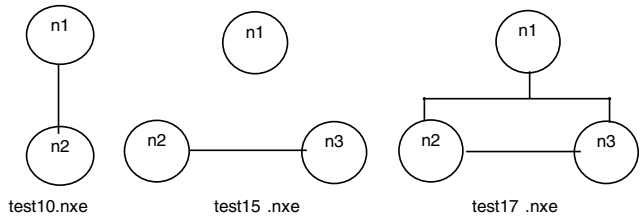

Fig. 5 Adding a node and a hyperedge to a small graph

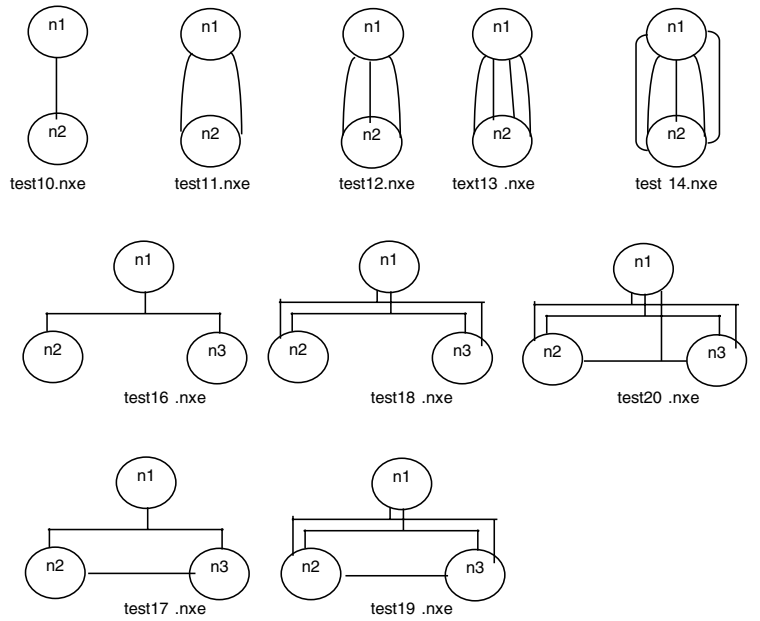

Fig. 6 Identical hyperedges do not add information

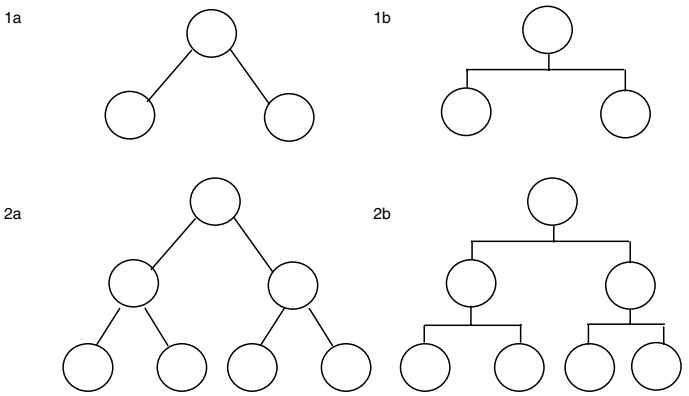

Fig. 7 Binary trees with ordinary edges vs. hyperedges

In summary, we have seen the following.

- Information theory metrics can differ from corresponding counting metrics.
- The information metrics are not sensitive to redundant hyperedges, because they do not add information in this model.
- Hypergraphs have less information than corresponding ordinary graphs.

Table 17 System-level measurements of trees with ordinary edges vs. hyperedges

	Ordinary edges		Hyperedges	
	Information	Count	Information	Count
System	1a		1b	
Size	6.0 bits	3 nodes	1.2 bits	3 nodes
Complexity	6.0 bits	2 edges	2.5 bits	1 hyperedge
System	2a		2b	
Size	21.0 bits	7 nodes	17.0 bits	7 nodes
Complexity	45.0 bits	6 edges	45.5 bits	3 hyperedges
System	3a		3b	
Size	49.5 bits	13 nodes	35.2 bits	13 nodes
Complexity	115.1 bits	12 edges	150.2 bits	4 hyperedges
System	4a		4b	
Size	30.0 bits	9 nodes	23.9 bits	9 nodes
Complexity	84.6 bits	10 edges	88.6 bits	3 hyperedges

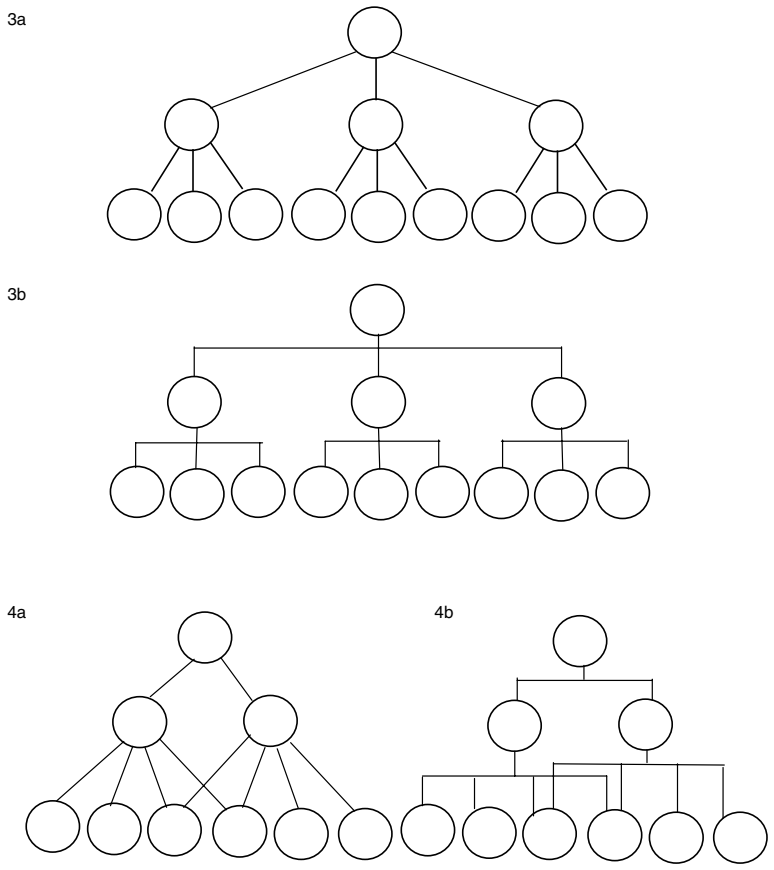

Fig. 8 Trees with ordinary edges vs. hyperedges

Table 18 Summary of physics program

Classes	3
Methods	14
Public variables	32
Language	C++
Function	Data manipulation

Table 19 Nodes × hyperedges table for the physics program

Module	Node	Hyperedges
env.	0	00000000000000000000000000000000
Lattice	get_periodic	10010011000011111011001000000111
	get_scale	01001100000000011100000111111001
	get_refvector	00110111110100010100111100010011
	get_parameter	00000100001010100000000000010000
	set_latticetype	00000000000000000100000000000000
	set_parameter	00000000000000000001000000000000
Element	get_mass	01100000000000000000000000000000
	get_name	01000000000000000000000000010000
	get_weight	00100000000000000100000000010000
	report	00100000000000000000010000000000
	setname	00000000000000001000000100101001
Atom	getpos	01100100000000010100000101010000
	setid	00000100001010000000000000010000
	setname	00000000000000000000000000000001

7.2 Use of public variables in a physics program

We analyzed a data manipulation program developed for a physics research project. This small program, summarized in Table 18, reads different data sets, manipulates the data, and writes the results to an output file. The source files were written in C++. Each class was defined as a module. The methods were represented by nodes, and each public variable was represented by a hyperedge. We derived a hypergraph from the relationships between public variables and the methods that use them. C++ system classes, their methods, and their public variables were excluded from the analysis, because we do not view them as part of the product that the programmer creates.

The primary C++ file (*.cpp) was preprocessed using the gcc compiler to generate a preprocessed (*.ii) file. This step included all header files (*.hpp) and subsidiary C++ files (*.cpp) into the resulting file (*.ii). The preprocessed file (*.ii) was parsed using the Datrix metric analyzer (Bell Canada, 2000b; Lapierre et al., 2001; Mayrand and Coallier, 1996), generating an abstract semantic graph (ASG, *.asg) (Bell Canada, 2000a). The ASG (*.asg) was input to our abstractor research tool, generating a nodes × hyperedges table (*.nxe). The *.nxe file was input to our measurement research tool for the metric calculations (Gottipati, 2003). Table 19 presents the nodes × hyperedges table for the program.

Table 20 presents the information theory-based and counting-based measurements. Table 20 shows that the module Element has medium information complexity, and module Atom has low information complexity. However, the counting complexity is about the same. Table 19 shows there are more connections in module Element than in module Atom,

Table 20 Measurements of the physics program

	Size		Complexity		Coupling	
	Information (bits)	Count (nodes)	Information (bits)	Count (hyperedges)	Information (bits)	Count (hyperedges)
System	54.7	14	366.3	32	341.2	15
Module:						
1. Lattice	23.4	6	172.6	32	157.1	15
2. Element	19.5	5	113.6	10	107.7	10
3. Atom	11.7	3	80.1	11	76.4	11

which helps explain the higher information complexity. The additional connections are not considered in counting complexity. The module Lattice contributes more than the other modules to all the system-level metrics. In this case study, if one uses a metric to order modules, the corresponding information metrics and the counting metrics generally result in the same order.

7.3 Use of public variables in a mathematical library

Development of the Parallel Mathematical Library Project (PMLP) (Birov et al., 1999) was a joint effort by Intel, Lawrence Livermore National Laboratory, the Russian Federal Nuclear Agency (VNIIEF), and the High Performance Computing Laboratory at the Mississippi State University. It is a parallel mathematical library suite for sparse matrices. PMLP includes sequential sparse basic linear algebra, parallel sparse matrix vector products, and sequential and parallel iterative solvers with Jacobi and incomplete LU preconditioners. PMLP consists of scalable libraries that combine the features of object-oriented design, sequential and parallel modes, etc. PMLP was developed in C++ using object-oriented techniques, such as template classes, generic programming, parametrized types, run-time polymorphism, compile-time polymorphism, and iterators. The major part of PMLP is coded as header files. Programs by users call its methods to perform various mathematical functions.

We identified several directories in PMLP version 4.0 containing related sets of files that can be considered "systems." Govindarajan studied three out of five systems (Govindarajan, 2004). In the interest of space, we present one system here, the Sequential Sparse Basic Linear Algebra system, SP_BLAS. The others had similar results. In this study, classes and templates were considered "modules." Methods were "nodes." Public variables were the "hyperedges," represented as columns in a nodes × hyperedges table. Table 21 presents a summary.

In object-oriented programming, variables are declared public in order to allow access from any class. "Intermodule hyperedges" are public variables accessed by at least two different classes. "Intramodule hyperedges" are public variables that are accessed only by methods in the same class; these variables would better be declared as private or protected. Complexity is defined in terms of all hyperedges, and coupling is defined in terms of

Table 21 Summary of SP_BLAS from PMLP

Classes/templates	95
Methods	681
Public variables	246
Language	C++
Function	Sparse linear algebra function library

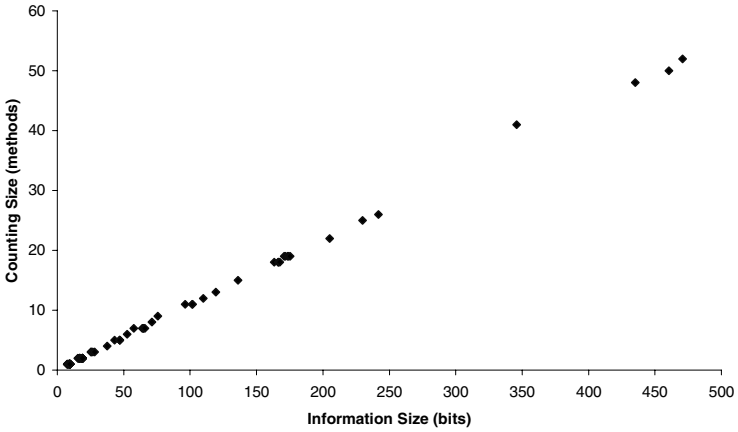

Fig. 9 Information module size *vs.* counting module size of SP_BLAS

intermodule hyperedges only. For an abstraction of software based on public variables, we expect all hyperedges to be intermodule. Therefore, complexity will be equal to coupling. This was largely the case in our analysis of PMLP. Consequently, we present a case study here that is limited to size and coupling.

Simple programs were produced by a test-program generator that was created for system testing. The resulting programs test various functions of SP_BLAS like matrix-element functions, matrix-matrix functions, matrix-vector, vector-element, and vector-vector functions. Test programs were a necessary vehicle for measurement, because PMLP functions are largely implemented by header files, but source code analyzers need a complete program to parse.

The test programs of the SP_BLAS were preprocessed using the gcc preprocessor, and the resultant file was used as an input to CPPX, a software analysis tool (University of Waterloo, 2004; Dean et al., 2001). CPPX generated an abstract semantic graph similar to that generated by Datrix in our case study above. The abstract semantic graph was analyzed for class-method-public variable relationships to generate a nodes × hyperedges table. There were 95 modules in SP_BLAS. Table 22 shows a sample fragment of the nodes × hyperedges table generated for SP_BLAS.

Information theory-based and counting-based measures were calculated for the nodes × hyperedges table. Table 23 shows the system-level information theory based metrics and counting-based metrics. Table 24 has summary statistics for the module-level measurements. Table 25 gives correlation between information theory-based module metrics to the corresponding counting-based metrics.

Information size is highly correlated with counting size. Figure 9 is a scatter plot comparing information module size with counting size. This visualizes their high correlation. When all the row patterns of the nodes in a module, m_k, are unique, then by definition, the information size of a module is proportional to the number of nodes in the module, n_k, namely, counting size (Govindarajan, 2004).

$$Size(m_k|\mathbf{S}) = n_k \left(-\log \frac{1}{n+1} \right) \tag{28}$$

Table 22 Sample nodes × hyperedges table for SP_BLAS

Module	Node	Row pattern
complex<float>	complex	11111110011000111110101100111011001111110011010 1...
'complex<long>	complex	0100010000100010000000110010100100000101001001000 10...
DenseRows<PREC>	Goto_first	0000000000100000000000000010000100000001000...
DenseRows<PREC>	Is_in_col_begin	0000000000000000100000000010101001 00001...
DNS<PREC>	Set_flag_sorted	1110010101110111110010110111111110011111101...
FormatOtherRow	FormatOtherRow	0000000000000000000001000000000000000000000001...
MatNotHermSymskewGen	'operator = '	1110000000000100000000000000000000000000...
Matrix<PR_SF_MT>	Remove_share	0010000100000000000000000000000000000...
MatrixPool<double>	Is_not_empty	0000000010000000000101001100000000000...
MatSymBase	MatSymBase	0000000000000000000000000000000000000000000000000000000000...
RefCount	Decr_ref_count	0000000000000000000000000000000000000000000010000100...
RefCount	Lock	0000000000000000000000001000000000000000001000010000...
SfCsrUser	SfCsrUser	1110010110010110000101001100010010000000100010100000 0...
sp_coo.h	Resize	0100010000000000000000000000000000000000000000000000000000...
sp_ell.h	Goto_first_in_row	0000000001000001010010000010000000100...
sp_list_err	sp_list_err	1110010000000010000000110000000000000010000 10010...
sp_mtherm.h	Get_element	1110010100010110000000110100010010000100101001000...
sp_mthermskew.h	Insert_element	1110010100000010000000000000000000000000010...
sp_mtsymskew.h	MatrixSymmetricSkew	1110010110010110000101111100010010000 10...
sp_skyupp.h	Goto_next_in_row	0000000000000000000000000000000000000000000000...
sp_sp_vec.h	'operator + = .	0000000000000000000000000000000000000000010000...
VectorBlocked<double>	'operator()'	1110010100010110000000110100010010000...
VectorRep<PR>	'operator()'	1110010110010110000101000100010010000000010 10...
VectorRep<PR>	VectorRep	1110010110010110000101001100010010000000010101 00...
...	...	...

Table 23 System measurements of SP_BLAS

	Size	Coupling
Information metric	6,174.4 bits	1,504,101.9 bits
Counting metric	681 methods	246 public variables

Table 24 Summary statistics for SP_BLAS module measurements

Module-level metric	Mean	Std dev.	Min.	Median	Max.
Information size (bits)	65.0	97.1	7.4	18.8	470.7
Information coupling (bits)	15, 832.6	24, 719.1	296.8	3, 960.7	129, 445.7
Counting size (methods)	7.2	10.7	1	2	52
Counting coupling (pubic variables)	78.6	68.7	1	57	239

Number of modules was 95.

Table 25 Correlation among SP_BLAS module measurements

	Counting size	Counting coupling
Information size	0.999	0.679
Information coupling	0.692	0.521

Number of modules was 95.

This accounts for the high correlation between information module size and counting size. All row patterns of many modules in SP_BLAS were unique, reflecting the usage patterns for global variables. Statistical modeling of SP_BLAS using both size metrics as independent variables would not be helpful due to the high correlation. The counting module size measure might be preferred, because the it is easier to collect than the information size.

Modules that had some nonunique row patterns are points that are not perfectly on the regression line. Such special cases, which have similar usage of global variables by multiple methods, may be of interest to practitioners.

Figure 10 shows a scatter plot representing information module coupling versus counting coupling measurements. Because it is a nonlinear relationship, the coupling measures have

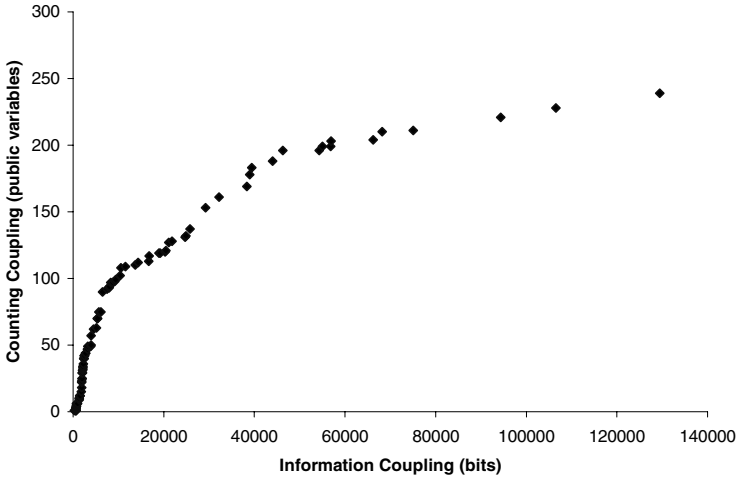

Fig. 10 Information module coupling vs. counting module coupling of SP_BLAS

moderate correlation values, as shown in Table 25. Table 25 also shows that the correlation between information module coupling and counting size is moderate, 0.692.

If each module has a unique pattern of usage of public variables, and each module also has a unique pattern in each subgraph, S_i, then information module coupling will be proportional to the number of nodes in the module, namely, counting coupling.

$$Coupling(m_k|\mathbf{S}) = n_k(n-1)\left(-\log\frac{1}{n+1}\right) \tag{29}$$

Some of the modules in this study fit these conditions, accounting for the moderate correlation.

8 Discussion

We have observed that the software-metrics literature tends to gloss over the distinction between forming an abstraction and measuring that abstraction, calling the entire process "definition of a software metric." This paper separates the abstraction process from the measurement method, allowing many aspects of software to be modeled by the same abstraction type which can then be measured in a unified manner.

Kitchenham et al. propose a framework for validating software metrics from a theoretical perspective, irrespective of a metric's utility (Kitchenham et al., 1995). Our proposed metrics fulfill the following criteria proposed by Kitchenham et al.

- Measurements are allowed to take different values.
- Multiple systems are allowed to have the same values.
- Each metric is a valid representation of its attribute, because it essentially conforms to the properties of Briand et al. (1996b), with the exception of module complexity.
- Each metric has an accepted unit of measure, namely, a *bit* which is widely used for information.
- All the metrics are compatible with the ratio scale-type (Briand et al., 1996a).
- Given a hypergraph, the measurement procedures are well defined.

Briand et al. (1996b) advocate ordinary graphs as a useful type of abstraction of software. Our work extends this notion to hypergraphs, which are well-suited to modeling some attributes. If a system's abstraction consists of one connected ordinary graph, $\mathbf{S}_c$, $n > 2$ (not considering an environment node), then the formula for the entropy of its system graph, S_c, is simplified, because each hyperedge has exactly two end points and each node's pattern is unique (Allen, 2002).

$$H(S_c) = \log(n+1) \tag{30}$$

In such systems, our metrics are highly correlated to simple functions of the number of nodes (Allen, 2002). Consequently, we suspect that the proposed information theory-based metrics will have value to practitioners when applied to abstractions of software suited to hypergraphs with varied numbers of connections, rather than ordinary graphs.

Counting features is the measurement method most often found in the software metrics literature. This research contributes an alternative approach that is not based on counting. Our focus is on measuring the amount of information in the abstraction. When features are independent, then counts and information can be highly correlated, and counts may be

preferable. When patterns of relationships are important, then information theory offers a way to measure them, as illustrated by our case study results.

The results of the case study of artificial hypergraphs illustrate how information theory-based metrics are sensitive to patterns of hyperedge connections in a way that corresponding counting metrics are not. However, our metrics are not sensitive to redundant hyperedges, and thus, our metrics do not model the idea of "strength of relationship." The results also show how hyperedges model sets of similar relationships more compactly than corresponding sets of individual ordinary edges. This could be valuable when relationships can be grouped in sets according to the underlying semantics.

The case studies of the physics program and PMLP represent scaling up to 14 methods and 681 methods respectively. In the PMLP case study, information module size was highly correlated to counting size, but not perfectly; exceptional cases may be of interest to software engineers. Information module coupling had a non-linear relationship with counting coupling, which we interpret to indicate that they measure different things.

9 Conclusions

This paper lays measurement foundations for future empirical research that will assess relationships between the amounts of information in software development artifacts and attributes of software quality and process. We build on proposals of Briand, Morasca, and Basili that graphs are a useful abstraction of software for measurement purposes, and that property sets are a practical way to give meaning to common terminology, such as size, length, complexity, coupling, and cohesion (Briand et al., 1996b).

We extend prior work to hypergraphs where a hyperedge signifies a relationship among a subset of nodes, in contrast to an ordinary edge representing a binary relationship between just two nodes. To measure the amount of information, we turn to information theory as the basis for calculating size, complexity, and coupling. For comparison, we also propose corresponding counting-based metrics.

Three exploratory case studies illustrate some of the distinctive features of information theory-based software metrics. The first case study looked at a set of small artificially generated graphs. The second case study examined relationships between methods and public variables in a single C++ program. The third case study measured relationships between methods and public variables in 95 C++ classes from the Parallel Mathematical Library Project (PMLP) (Birov et al., 1999). We found some situations where information theory-based metrics at the module level are simply proportional to counting-based metrics. In general, the case studies found that information theory-based software metrics distinguish various configurations of hypergraph connections in a way that counting metrics do not. Future work may show that this is relevant to software engineering quality and process.

Future work will evaluate the formal properties of these measures in more depth, and will evaluate their usefulness in the context of full-scale real-world software products. Similar extensions to length and cohesion are also expected in the future, as well as extensions to directed graphs.

Information theory addresses the amount of information when compactly encoded. In future research, cognitive-science experiments will be necessary to determine the extent that perceived information corresponds to compactly encoded information. This in turn, will indicate whether information theory provides a good basis for modeling software as perceived by programmers.

Acknowledgments This work was supported in part by grant CCR-0098024 from the National Science Foundation. We thank Bell Canada for an academic license to use Datrix, a software measurement tool. We thank the Software Architecture Group of the University of Waterloo for providing the open-source tool *cppx*. We thank Shiva Juluru for providing the physics data manipulation program's source code. We thank Anthony Skjellum for providing PMLP source code. We thank Yoginder Dandass and Archana Chilukuri for help with PMLP measurement. We thank the Empirical Software Engineering research group at Mississippi State University for helpful discussions. We thank the anonymous reviewers for their helpful suggestions which significantly strengthened the paper.

References

Abd-El-Hafiz, S.K. 2001. Entropies as measures of software information. In: Proceedings IEEE International Conference on Software Maintenance, Florence, Italy. IEEE Computer Society, pp. 110–117.

Allen, E.B. 1995. Information theory and software measurement. PhD thesis, Florida Atlantic University, Boca Raton, Florida. Advised by Taghi M. Khoshgoftaar.

Allen, E.B. 2002. Measuring graph abstractions of software: An information-theory approach. In: Proceedings: Eighth IEEE Symposium on Software Metrics, Ottawa, Canada. IEEE Computer Society, pp. 182–193.

Allen, E.B., Khoshgoftaar, T.M. 1999. Measuring coupling and cohesion: An information-theory approach. In: Proceedings of the Sixth International Software Metrics Symposium, Boca Raton, Florida. IEEE Computer Society, pp. 119–127.

Allen, E.B., Khoshgoftaar, T.M., Chen, Y. 2001. Measuring coupling and cohesion of software modules: An information-theory approach. In: Proceedings: Seventh International Software Metrics Symposium, London, England. IEEE Computer Society, pp. 124–134.

Andersson, C., Thelin, T., Runeson, P., Dzamashvili, N. 2003. An experimental evaluation of inspection and testing for detection of design faults. In: Proceedings: 2003. International Symposium on Empirical Software Engineering, Rome, Italy. IEEE Computer Society, pp. 174–184.

Bansiya, J., Davis, C.G., Etzkorn, L. 1999. An entropy based complexity measure for object-oriented designs. Theory and Practice of Object Systems **5**(2):1–9.

Bell Canada 2000a. Datrix Abstract Semantic Graph Reference-Manual (Version 1.4).

Bell Canada 2000b. Datrix Metric Reference Manual. Montreal, Quebec, Canada, version 4.0 edition. For Datrix version 3.6.9.

Birov, L., Prokofiev, A., Bartenev, Y., Vargin, A., Purkayastha, A., Skjellum, A., Dandass, Y., Erzunov, V., Shanikova, E., Ovechkin, V., Bangalore, P., Shuvalov, E., Orlov, N.F.A., Egorov, S. 1999. The Parallel Mathematical Libraries Project (PMLP): Overview, design innovations, and preliminary results. In: Proceedings of the Fifth International Conference on Parallel Computing Technologies.

Briand, L.C., Daly, J.W., Wüst, J. 1997a. A unified framework for cohesion measurement in object-oriented systems. In: Proceedings of the Fourth International Symposium on Software Metrics, Albuquerque, New Mexico. IEEE Computer Society, pp. 43–53.

Briand, L.C., Daly, J.W., Wüst, J.K. 1999. A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering **25**(1):91–121.

Briand, L.C., El Emam, K., Morasca, S. 1996a. On the application of measurement theory in software engineering. Empirical Software Engineering: An International Journal **1**(1):61–88. (See Briand et al., 1997b; Zuse, 1997a).

Briand, L.C., El Emam, K., Morasca, S. 1997b. Reply to Comments to the paper: Briand, El Emam, Morasca: On the application of measurement theory in software engineering. Empirical Software Engineering: An International Journal **2**(3):317–322. (See Briand et al., 1996a; Zuse, 1997a).

Briand, L.C., Morasca, S., Basili, V.R. 1996b. Property-based software engineering measurement. IEEE Transactions on Software Engineering **22**(1):68–85. See comments in Briand et al. (1997c), Poels and Dedene (1997), Zuse (1997c).

Briand, L.C., Morasca, S., Basili, V.R. 1997c. Response to: Comments on Property-based software engineering measurement: Refining the additivity properties. IEEE Transactions on Software Engineering **23**(3):196–197. (See Briand et al., 1996b; Poels and Dedene, 1997).

Chaitin, G.J. 1966. On the length of programs for computing finite binary sequences. Journal of the Association for Computing Machinery **13**(4):547–569.

Chaitin, G.J. 1975. A theory of program size formally identical to information theory. Journal of the Association for Computing Machinery **22**(3):329–340.

Chapin, N. 2002. Entropy-metric for systems with COTS software. In: Proceedings: Eighth IEEE Symposium on Software Metrics, Ottawa, Canada. IEEE Computer Society, pp. 173–181.

Chen, Y. 2000. Measurement of coupling and cohesion of software. Master's thesis, Florida Atlantic University, Boca Raton, Florida. Advised by Taghi M. Khoshgoftaar.

Chidamber, S.R., Kemerer, C.F. 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering **20**(6):476–493.

Cover, T.M., Thomas, J.A. 1991. Elements of Information Theory. John Wiley & Sons, New York.

Davis, J.S., LeBlanc, R.J. 1988. A study of the applicability of complexity measures. IEEE Transactions on Software Engineering **14**(9):1366–1372.

Dean, T., Malton, A., Holt, R. 2001. Union schemas as the basis for a C++ extractor. In: Proceedings: Working Conference on Reverse Engineering, Stuttgart, Germany.

El Emam, K., Benlarbi, S., Goel, N., Rai, S.N. 2001. The confounding effect of class size on the validity of object-oriented metrics. IEEE Transactions on Software Engineering **27**(7):630–650. (See Evanco, 2003).

Evanco, W.M. 2003. Comments on 'The confounding effect of class size on the validity of object-oriented metrics'. IEEE Transactions on Software Engineering **29**(7):670–672. (See El Emam et al., 2001).

Fenton, N.E., Pfleeger, S.L. 1997. Software Metrics: A Rigorous and Practical Approach, 2nd edn. PWS Publishing, London.

Gottipati, S. 2003. Empirical validation of the usefulness of information theory-based software metrics. Master's thesis, Mississippi State University, Mississippi State, Mississippi. Advised by Edward B. Allen.

Govindarajan, R. 2004. An empirical validation of information theory-based software metrics in comparison to counting-based metrics: A case study approach. Master's thesis, Mississippi State University, Mississippi State, Mississippi. Advised by Edward B. Allen.

Hatton, L. 1997. Reexamining the fault density-component size connection. IEEE Software **14**(2):89–97.

Hilgard, E.R., Atkinson, R.C., Atkinson, R.L. 1971. Introduction to Psychology. Harcourt Brace Jovanovich, New York.

Khoshgoftaar, T.M., Allen, E.B. 1994. Applications of information theory to software engineering measurement. Software Quality Journal **3**(2):79–103.

Kim, K., Shin, Y., Wu, C. 1995. Complexity measures for object oriented program based on the entropy. In: Proceedings: 1995 Asia Pacific Software Engineering Conference, Brisbane, Australia. IEEE Computer Society, pp. 127–136.

Kitchenham, B.A., Pfleeger, S.L., Fenton, N.E. 1995. Towards a framework for software measurement validation. IEEE Transactions on Software Engineering **21**(12):929–944. (See comments in Kitchenham et al. 1997, Morasca et al., 1997).

Kitchenham, B.A., Pfleeger, S.L., Fenton, N.E. 1997. Reply to: Comments on 'Towards a framework for software measurement validation'. IEEE Transactions on Software Engineering **23**(3):189. (See Kitchenham et al., 1995; Morasca et al., 1997; Weyuker, 1988).

Kolmogorov, A.N. 1965. Three approaches for defining the concept of information quantity. Problems in Information Transmission **1**(1):1–7.

Kolmogorov, A.N. 1968. Logical basis for information theory and probability theory. IEEE Transactions on Information Theory **IT-14**(5):662–664.

Lapierre, S., Laguë, B., Leduc, C. 2001. Datrix source code model and its interchange format: Lessons learned and considerations for future work. ACM SIGSOFT Software Engineering Notes **26**(1):53–60.

Lew, K.S., Dillon, T.S., Forward, K.E. 1988. Software complexity and its impact on software reliability. IEEE Transactions on Software Engineering **14**(11):1645–1655.

Li, M., Vitányi, P.M.B. 1988. Two decades of applied Kolmogorov complexity. In: Proceedings of the Third Annual Structure in Complexity Theory Conference, Washington, DC, pp. 80–101.

Mayrand, J., Coallier, F. 1996. System acquisition based on software product assessment. In: Proceedings of the Eighteenth International Conference on Software Engineering, Berlin. IEEE Computer Society, pp. 210–219.

McCabe, T.J. 1976. A complexity measure. IEEE Transactions on Software Engineering **SE-2**(4):308–320.

Miller, G.A. 1956. The magical number seven plus or minus two: Some limits on our capacity for processing information. Psychological Review **63**(2):81–97.

Mohanty, S.N. 1979 Models and measurements for quality assessment of software. Computing Surveys **11**(3):251–275.

Mohanty, S.N. 1981. Entropy metrics for software design evaluation. Journal of Systems and Software **2**:39–46.

Morasca, S., Briand, L.C. 1997. Towards a theoretical framework for measuring software attributes. In: Proceedings of the Fourth International Symposium on Software Metrics, Albuquerque, New Mexico, IEEE Computer Society, pp. 119–126.

Morasca, S., Briand, L.C., Basili, V.R., Weyuker, E.J., Zelkowitz, M.V. 1997. Comments on 'Towards a framework for software measurement validation'. IEEE Transactions on Software Engineering **23**(3):187–188. (See Kitchenham et al., 1995; Weyuker, 1988).

Munson, J.C., Khoshgoftaar, T.M. 1989. The dimensionality of program complexity. In: Proceedings of the Eleventh International Conference on Software Engineering, Pittsburgh, Pennsylvania. IEEE Computer Society, pp. 245–253.

Oviedo, E.I. 1980. Control flow, data flow and program complexity. In: Proceedings: The IEEE Computer Society's Fourth International Computer Software and Applications Conference, Chicago, Illinois. IEEE Computer Society, pp. 146–152.

Poels, G., Dedene, G. 1997 Comments on 'Property-based software engineering measurement': Refining the additivity properties. IEEE Transactions on Software Engineering **23**(3):190–195. (See Briand et al., 1996b).

Runeson, P., Andersson, C., Thelin, T., Andrews, A., Berling, T. 2006. What do we know about defect detection methods? IEEE Software **23**(3):82–90.

Schütt, D. 1977. On a hypergraph oriented measure for applied computer science. In Digest of Papers: COMPCON 77 Fall, Washington, DC. IEEE Computer Society, pp. 295–296, Abstract only.

Shannon, C.E., Weaver, W. 1949. The Mathematical Theory of Communication. University of Illinois Press, Urbana, Illinois.

Shereshevshky, M., Ammari, H., Gradetsky, N., Mili, A., Ammar, H.H. 2001. Information theoretic metrics for software architecture. In: Proceedings 25th Annual International Computer Software and Applications Conference, Chicago. IEEE Computer Society, pp. 151–157.

Solomonoff, R.J. 1964. A formal theory of inductive inference, part 1 and part 2. Information and Control **7**:1–22, 224–254.

University of Waterloo 2004. CPPX: Open source C++ fact extractor. http://swag.uwaterloo.ca/∼cppx. (Current July 7, 2006).

van Emden, M.H. 1970. Hierarchical decomposition of complexity. Machine Intelligence **5**:361–380. (See also van Emden, 1971 for details).

van Emden, M.H. 1971. An Analysis of Complexity. Number 35 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam.

Visaggio, G. 1997. Structural information as a quality metric in software systems organization. In: Proceedings International Conference on Software Maintenance, Bari, Italy. IEEE Computer Society, pp. 92–99.

Watanabe, S. 1960. Information theoretical analysis of multivariate correlation. IBM Journal of Research and Development **4**(1):66–82.

Weyuker, E.J. 1988. Evaluating software complexity measures. IEEE Transactions on Software Engineering **14**(9):1357–1365.

Zuse, H. 1997a. Comments to the paper: Briand, Emam, Morasca: On the application of measurement theory in software engineering. Empirical Software Engineering: An International Journal **2**(3):313–316. (See Briand et al., 1996a, 1997b).

Zuse, H. 1997b. A Framework for Software Measurement. Walter de Gruyter and Co., Berlin.

Zuse, H. 1997c. Reply to: 'Property-based software engineering measurement'. IEEE Transactions on Software Engineering **23**(8):533. (See Briand et al., 1996b).

Edward B. Allen is currently an Associate Professor at Mississippi State University. He received his B.S. degree in engineering from Brown University, his M.S. degree in systems engineering from the University of Pennsylania, and his Ph.D. degree in computer science from Florida Atlantic University in 1995. Prior to earning his Ph.D. degree, he worked for 20 years in industry performing systems engineering and software engineering on military systems and corporate data processing systems. His research interests are software metrics, verification of software for critical systems, and other areas of software engineering.

Sampath Gottipati is currently a programmer for Technation Software Consulting. He received his B.E. in computer science degree from the University of Madras and his M.S. degree in computer science from Mississippi State Univerisity in 2003. His research interest is software engineering.

Rajiv Govindarajan is currently a programmer for Peri Software Solutions. He received his B.E. in computer science degree from the University of Madras and his M.S. degree in computer science from Mississippi State University in 2004. His research interest is software engineering.