Property-based Software Engineering Measurement

Lionel Briand
CRIM
1801 McGill College Avenue
Montréal (Quebec), H3A 2N4
Canada
Lionel.Briand@crim.ca

Sandro Morasca
Dip. di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci 32
I-20133 Milano, Italy
morasca@elet.polimi.it

Victor R. Basili
Computer Science Department
University of Maryland
College Park, MD 20742
basili@cs.umd.edu

Abstract

Little theory exists in the field of software system measurement. Concepts such as complexity, coupling, cohesion or even size are very often subject to interpretation and appear to have inconsistent definitions in the literature. As a consequence, there is little guidance provided to the analyst attempting to define proper measures for specific problems. Many controversies in the literature are simply misunderstandings and stem from the fact that some people talk about different measurement concepts under the same label (complexity is the most common case).

There is a need to define unambiguously the most important measurement concepts used in the measurement of software products. One way of doing so is to define precisely what mathematical properties characterize these concepts, regardless of the specific software artifacts to which these concepts are applied. Such a mathematical framework could generate a consensus in the software engineering community and provide a means for better communication among researchers, better guidelines for analysts, and better evaluation methods for commercial static analyzers for practitioners.

In this paper, we propose a mathematical framework which is generic, because it is not specific to any particular software artifact, and rigorous, because it is based on precise mathematical concepts. This framework defines several important measurement concepts (size, length, complexity, cohesion, coupling). It does not intend to be complete or fully objective; other frameworks could have been proposed and different choices could have been made. However, we believe that the formalisms and properties we introduce are convenient and intuitive. In addition, we have reviewed the literature on this subject and compared it with our work. This framework contributes constructively to a firmer theoretical ground of software measurement.

1. Introduction

Many concepts have been introduced through the years to define the characteristics of the artifacts produced during the software process. For instance, one speaks of size and complexity of software specification, design, and code, or cohesion and coupling of a software design or code. Several techniques have been introduced, with the goal of producing software which is better with respect to these concepts. As an example, Parnas [P72] design principles attempt to decrease coupling between modules, and increase cohesion within modules. These concepts are used as a guide to choose among alternative techniques or artifacts. For instance, a technique may be preferred over another because it yields artifacts that are less complex; an artifact may be preferred over another because it is less complex. In turn, lower complexity is believed to provide advantages such as lower maintenance time and cost. This shows the importance of a clear and unambiguous understanding of what these concepts actually mean, to make choices on more objective bases. The

This work was supported in part by NASA grant NSG–5123, UMIACS, NSF grant 01-5-24845, MURST, and CNR. This Tecnical Report is also available as Internal Report 94.078, Politecnico di Milano, Dipartimento di Elettronica e Informazione.

definition of relevant concepts (i.e., classes of software characterization measures) is the first step towards quantitative assessment of software artifacts and techniques, which is needed to assess risk and find optimal trade-offs between software quality, schedule and cost of development.

To capture these concepts in a quantitative fashion, hundreds of software measures have been defined in the literature. However, the vast majority of these measures did not survive the proposal phase, and did not manage to get accepted in the academic or industrial worlds. One reason for this is the fact that they have not been built using a clearly defined process for defining software measures. As we propose in [BMB94(b)], such a process should be driven by clearly identified measurement goals and knowledge of the software process. One of its crucial activities is the precise definition of relevant concepts, necessary to lay down a rigorous framework for software engineering measures and to define meaningful and well-founded software measures. The theoretical soundness of a measure, i.e., the fact that it really measures the software characteristic it is supposed to measure, is an obvious prerequisite for its acceptability and use. The exploratory process of looking for correlations is not an acceptable scientific validation process in itself if it is not accompanied by a solid theory to support it. Unfortunately, new software measures are very often defined to capture elusive concepts such as complexity, cohesion, coupling, connectivity, etc. (Only size can be thought to be reasonably well understood.) Thus, it is impossible to assess the theoretical soundness of newly proposed measures, and the acceptance of a new measure is mostly a matter of belief.

To this end, several proposals have appeared in the literature [LJS91, TZ92, W88] in recent years to provide desirable properties for software measures. These works (especially [W88]) have been used to "validate" existing and newly proposed software measures. Surprisingly, whenever a new measure which was proposed as a software complexity measure did not satisfy the set of properties against which it was checked, several authors failed to conclude that their measure was not a software complexity measure, e.g., [CK94, H92]. Instead, they concluded that their measure was a complexity measure that does not satisfy that set of properties for complexity measures. What they actually did was provide an absolute definition of a software complexity measure and check whether the properties were consistent with respect to the measure, i.e., check the properties against their own measure.

This situation would be unacceptable in other engineering or mathematical fields. For instance, suppose that one defines a new measure, claiming it is a distance measure. Suppose also that that measure fails to satisfy the triangle inequality, which is the characterizing property of distance measures. The natural conclusion would be to realize that that is not a distance measure, rather than to say that it is a distance measure that does not satisfy the conditions for a distance measure. However, it is true that none of the sets of properties proposed so far has reached so wide an acceptance to be considered "the" right set of necessary properties for complexity. It is our position that this odd situation is due to the fact that there are several different concepts that are still covered by the same word: complexity.

Within the set of commonly mentioned software characteristics, size and complexity are the ones that have received the widest attention. However, the majority of authors have been inclined to believe that a measure captures either size or complexity, as if, besides size, all other concepts related to software characteristics could be grouped under the unique name of complexity. Sometimes, even size has been considered as a particular kind of complexity measure.

Actually, these concepts capture different software characteristics, and, until they are clearly separated and their similarities and differences clearly studied, it will be impossible to reach any kind of consensus on the properties that characterize each concept relevant to the definition of software measures. The goal of this paper is to lay down the basis for a discussion on this subject, by providing properties for a—partial—set of measurement concepts that are relevant for the definition of software measures. Many of the measure properties proposed in the literature are generic in the sense that they do not characterize specific measurement concepts but are relevant to all syntactically-based measures (see [S92, TZ92, W88]). In this paper, we want to focus on properties that differentiate measurement concepts such as size, complexity, coupling, etc. Thus, we want to identify and clarify the essential properties behind these concepts that are commonplace in software engineering and form important classes of measures. Thus, researchers will be able to validate their new measures by checking properties specifically relevant to the class (or concept) they belong to (e.g., size should be additive). *By no means should these properties be regarded as* *the unique set of properties that can be possibly defined for a given concept.* Rather, we want to provide a theoretically sound and convenient solution for differentiating a set of well known concepts and check their analogies and conflicts**.** Possible applications of such a framework are to guide researchers in their search for new measures and help practitioners evaluate the adequacy of measures provided by commercial tools.

We also believe that the investigation of measures should also address artifacts produced in the software process other than code. It is commonly believed that the early software process phases are the most important ones, since the rest of the development depends on the artifacts they produce. Oftentimes, the concepts (e.g., size, complexity, cohesion, coupling) which are believed relevant with respect to code are also relevant for other artifacts. To this end, the properties we propose will be general enough to be applicable to a wide set of artifacts.

The paper is organized as follows. In Section 2, we introduce the basic definitions of our framework. Section 3 provides a set of properties that characterize and formalize intuitively relevant measurement concepts: size, length, complexity, cohesion, coupling. We also discuss the relationships and differences between the different concepts. Some of the best-known measures are used as examples to illustrate our points. Section 4 contains comparisons and discussions regarding the set of properties for complexity measures defined in the paper and in the literature. The conclusions and directions for future work come in Section 5.

2. Basic Definitions

Before introducing the necessary properties for the set of concepts we intend to study, we provide basic definitions related to the objects of study (to which these concepts can be applied), e.g., size and complexity of *what*?

Systems and modules

Two of the concepts we will investigate, namely, size (Section 3.1) and complexity (Section 3.3) are related to systems, in general, i.e., one can speak about the size of a system and the complexity of a system. We also introduce a new concept, length (Section 3.2), which is related to systems. In our general framework—recall that we want these properties to be as independent as possible of any product abstraction—, a system is characterized by its elements and the relationships between them. Thus, we do not reduce the number of possible system representations, as elements and relationships can be defined according to needs.

Definition 1: Representation of Systems and Modules
A *system* S will be represented as a pair $<E,R>$, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements.

Given a system $S = <E,R>$, a system $m = <E_m,R_m>$ is a *module* of S if and only if $E_m \subseteq E$, $R_m \subseteq E \times E$, and $R_m \subseteq R$. As an example, E can be defined as the set of code statements and R as the set of control flows from one statement to another. A module m may be a code segment or a subprogram.

The elements of a module are connected to the elements of the rest of the system by incoming and outgoing relationships. The set InputR(m) of relationships from elements outside module $m = <E_m,R_m>$ to those of module m is defined as

$$\mathrm{InputR}(m) = \{<e_1,e_2> \in R \mid e_2 \in E_m \ \textbf{and}\ e_1 \in E - E_m\}$$

The set OutputR(m) of relationships from the elements of a module $m = <E_m,R_m>$ to those of the rest of the system is defined as

$$\mathrm{OutputR}(m) = \{<e_1,e_2> \in R \mid e_1 \in E_m \ \textbf{and}\ e_2 \in E - E_m\}$$

◊

We now introduce inclusion, union, intersection operations for modules and the definitions of empty and disjoint modules, which will be used often in the remainder of the paper. For notational convenience, they will be denoted by extending the usual set-theoretic notation. We will illustrate these operations by means of the system $S = \langle E,R \rangle$ represented in Figure 1, where $E = \{a,b,c,d,e,f,g,h,i,j,k,l,m\}$ and $R = \{\langle b,a\rangle,\langle b,f\rangle,\langle c,b\rangle,\langle c,d\rangle,\langle c,g\rangle,\langle d,f\rangle,\langle e,g\rangle,\langle f,i\rangle, \langle f,k\rangle,\langle g,m\rangle,\langle h,a\rangle,\langle h,i\rangle,\langle i,j\rangle,\langle k,j\rangle,\langle k,l\rangle\}$. We will consider the following modules

- $m_1 = \langle E_{m1},R_{m1}\rangle = \langle\{a,b,f,i,j,k\},\{\langle b,a\rangle,\langle b,f\rangle,\langle f,i\rangle,\langle f,k\rangle,\langle i,j\rangle,\langle k,j\rangle\}$ (area filled with)
- $m_2 = \langle E_{m2},R_{m2}\rangle = \langle\{f,j,k\},\{\langle f,k\rangle,\langle k,j\rangle\}$ (area filled with)
- $m_3 = \langle E_{m3},R_{m3}\rangle = \langle\{c,d,e,f,g,j,k,m\},\{\langle c,d\rangle,\langle c,g\rangle,\langle d,f\rangle,\langle e,g\rangle,\langle f,k\rangle,\langle g,m\rangle, \langle k,j\rangle\}\rangle$ (area filled with)
- $m_4 = \langle E_{m4},R_{m4}\rangle = \langle\{d,e,g\},\{\langle e,g\rangle\}\rangle$ (area filled with)

Inclusion. Module $m_1 = \langle E_{m1},R_{m1}\rangle$ is said to be included in module $m_2 = \langle E_{m2},R_{m2}\rangle$ (notation: $m_1 \subseteq m_2$) if $E_{m1} \subseteq E_{m2}$ and $R_{m1} \subseteq R_{m2}$. In Figure 1, $m_4 \subseteq m_3$.

Union. The union of modules $m_1 = \langle E_{m1},R_{m1}\rangle$ and $m_2 = \langle E_{m2},R_{m2}\rangle$ (notation: $m_1 \cup m_2$) is the module $\langle E_{m1} \cup E_{m2},R_{m1} \cup R_{m2}\rangle$. In Figure 1, the union of modules m_1 and m_3 is module $m_{13} = \langle\{a,b,c,d,e,f,g,i,j,k,m\}, \{\langle b,a\rangle,\langle b,f\rangle,\langle c,d\rangle,\langle c,g\rangle,\langle d,f\rangle,\langle e,g\rangle,\langle f,i\rangle, \langle f,k\rangle,\langle g,m\rangle,\langle i,j\rangle,\langle k,j\rangle\}$ (area filled with or or).

Intersection. The intersection of modules $m_1 = \langle E_{m1},R_{m1}\rangle$ and $m_2 = \langle E_{m2}, R_{m2}\rangle$ (notation: $m_1 \cap m_2$) is the module $\langle E_{m1} \cap E_{m2},R_{m1} \cap R_{m2}\rangle$. In Figure 1, $m_2 = m_1 \cap m_3$.

Empty module. Module $\langle\varnothing,\varnothing\rangle$ (denoted by $\varnothing$) is the empty module.

Disjoint modules. Modules m_1 and m_2 are said to be disjoint if $m_1 \cap m_2 = \varnothing$. In Figure 1, $m_1 \cap m_4 = \varnothing$.

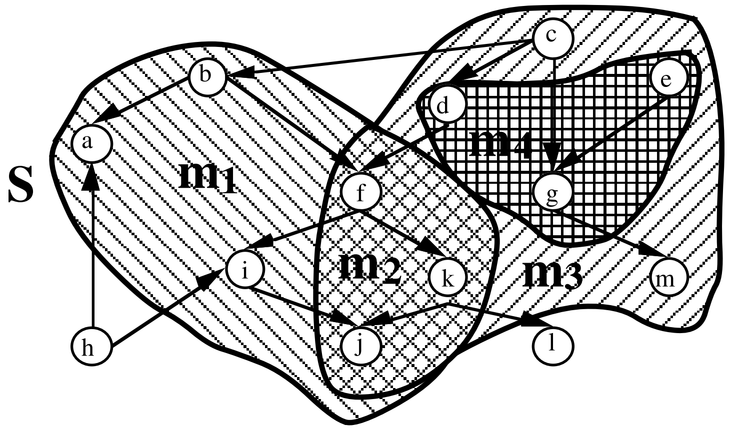

Figure 1. Operations on modules.

Since in this framework modules are just subsystems, all systems can theoretically be decomposed into modules. The definition of a module for a particular measure in a specific context is just a matter of convenience and programming environment (e.g., language) constraints.

Modular systems

The other two concepts we will investigate, cohesion (Section 3.4) and coupling (Section 3.5), are meaningful only with reference to systems that are provided with a modular decomposition, i.e., one can speak about cohesion and coupling of a whole system only if it is structured into modules. One can also speak about cohesion and coupling of a single module within a whole system.

Definition 2: Representation of Modular Systems
The 3-tuple MS = <E,R,M> represents a modular system if S = <E,R> is a system according to Definition 1, and M is a collection of modules of S such that

$\forall e \in E \ (\exists m \in M \ (m = \langle E_m, R_m \rangle$ **and** $e \in E_m))$ **and**

$\forall m_1, m_2 \in M \ (m_1 = \langle E_{m1}, R_{m1} \rangle$ **and** $m_2 = \langle E_{m2}, R_{m2} \rangle$ **and** $E_{m1} \cap E_{m2} = \emptyset)$

i.e, the set of elements E of MS is partitioned into the sets of elements of the modules.

We denote the union of all the R_m's as IR. It is the set of *intra-module relationships*. Since the modules are disjoint, the union of all OutputR(m)'s is equal to the union of all InputR(m)'s, which is equal to R-IR. It is the set of *inter-module relationships*.

◊

As an example, E can be the set of all declarations of a set of Ada modules, R the set of dependencies between them, and M the set of Ada modules.

Figure 2 shows a modular system MS = <E,R,M>, obtained by partitioning the set of elements of the system in Figure 1 in a different way. In this modular system, E and R are the same as in system S in Figure 1, and $M = \{m_1, m_2, m_3\}$. Besides, IR = {<b,a>,<c,d>,<c,g>, <e,g>,<f,i>,<f,k>,<g,m>,<h,a>,<i,j>,<k,j>,<k,l>}.

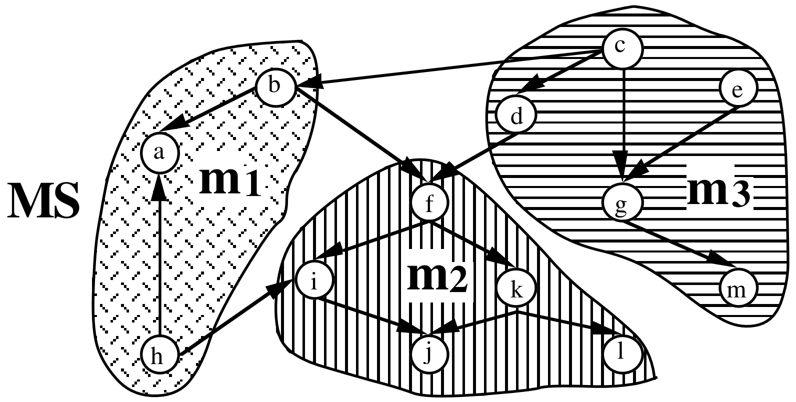

Figure 2. A modular system.

It should be noted that some measures do not take into account the modular structure of a system. As already mentioned, our concepts of size and complexity (defined in Sections 3.1 and 3.3) are such examples, i.e., in a modular system MS = <E,R,M>, one computes size and complexity of the system S = <E,R>, and M is not considered.

We have defined concept properties using a graph-theoretic approach to allow us to be general and precise. It is general because our properties are defined so that no restriction applies to the definition of vertices and arcs. Many well known product abstractions fit this framework, e.g., data dependency graphs, definition-use graphs, control flow graphs, USES graphs, Is_Component_of graphs, etc. It is precise because, based on a well defined formalism, all the concepts used can be mathematically defined, e.g., system, module, modular system, and so can the properties presented in the next section.

3. Concepts of Measurement and Properties

It should be noted that the concepts defined below are to some extent subjective. However, we wish to assign them intuitive and convenient properties. We consider these properties necessary but not sufficient because they do not guarantee that the measures for which they hold are useful or even make sense. On the other hand, these properties will constrain the search for measures and therefore make the measure definition process more rigorous and less exploratory [BMB94(b)]. Several relevant concepts are studied: size, length, complexity, cohesion, and coupling. They do not represent an exhaustive list but a starting point for discussion that should eventually lead to a standard definition set in the software engineering community.

3.1. Size

Motivation

Intuitively, size is recognized as being an important measurement concept. According to our framework, size cannot be negative (property Size.1), and we expect it to be null when a system does not contain any elements (property Size.2). When modules do not have elements in common, we expect size to be additive (property Size.3).

Definition 3: Size
The size of a system S is a function Size(S) that is characterized by the following properties Size.1 - Size.3.

◊

Property *Size.1*: Non-negativity
The size of a system $S = \langle E,R \rangle$ is non-negative

$$\mathrm{Size}(S) \geq 0 \qquad \text{(Size.I)}$$

◊

Property *Size.2*: Null Value
The size of a system $S = \langle E,R \rangle$ is null if E is empty

$$E = \varnothing \Rightarrow \mathrm{Size}(S) = 0 \qquad \text{(Size.II)}$$

◊

Property *Size.3*: Module Additivity
The size of a system $S = \langle E,R \rangle$ is equal to the sum of the sizes of two of its modules $m_1 = \langle E_{m1},R_{m1} \rangle$ and $m_2 = \langle E_{m2},R_{m2} \rangle$ such that any element of S is an element of either m_1 or m_2

$$(m_1 \subseteq S \ \textbf{and}\ m_2 \subseteq S \ \textbf{and}\ E = E_{m1} \cup E_{m2} \ \textbf{and}\ E_{m1} \cap E_{m2} = \varnothing)$$
$$\Rightarrow \mathrm{Size}(S) = \mathrm{Size}(m_1) + \mathrm{Size}(m_2) \qquad \text{(Size.III)}$$

◊

For instance, the size of the system in Figure 2 is the sum of the sizes of its three modules m_1,m_2,m_3.

The last property Size.3 provides the means to compute the size of a system $S = \langle E,R \rangle$ from the knowledge of the size of its—disjoint—modules $m_e = \langle \{e\},R_e \rangle$ whose set of elements is composed of a different element e of E[1].

[1]For each m_e, it is either $R_e = \varnothing$ or $R_e = \{\langle e,e \rangle\}$.

$$\mathrm{Size}(S) = \sum_{e \in E} \mathrm{Size}(m_e) \qquad \text{(Size.IV)}$$

Therefore, adding elements to a system cannot decrease its size (*size monotonicity property*)

$$(S' = \langle E',R' \rangle \textbf{ and } S'' = \langle E'',R'' \rangle \textbf{ and } E' \subseteq E'') \Rightarrow \mathrm{Size}(S') \leq \mathrm{Size}(S'') \qquad \text{(Size.V)}$$

From the above properties Size.1 - Size.3, it follows that the size of a system $S = \langle E,R \rangle$ is not greater than the sum of the sizes of any pair of its modules $m_1 = \langle E_{m1},R_{m1} \rangle$ and $m_2 = \langle E_{m2},R_{m2} \rangle$, such that any element of S is an element of m_1, or m_2, or both, i.e.,

$$(m_1 \subseteq S \textbf{ and } m_2 \subseteq S \textbf{ and } E = E_{m1} \cup E_{m2}) \Rightarrow \mathrm{Size}(S) \leq \mathrm{Size}(m_1) + \mathrm{Size}(m_2) \qquad \text{(Size.VI)}$$

The size of a system built by merging such modules cannot be greater than the sum of the sizes of the modules, due to the presence of common elements (e.g., lines of code, operators, class methods).

Properties Size.1 - Size.3 hold when applying the admissible transformation of the ratio scale [F91]. Therefore, there is no contradiction between our concept of size and the definition of size measures on a ratio scale.

Examples and counterexamples of size measures

Several measures introduced in the literature can be classified as size measures, according to our properties Size.1 - Size.3. With reference to code measures, we have: LOC, #Statements, #Modules, #Procedures, Halstead's Length [H77], #Occurrences of Operators, #Occurrences of Operands, #Unique Operators, #Unique Operands. In each of the above cases, the representation of a program as a system is quite straightforward. Each counted entity is an element, and the relationship between elements is just the sequential relationship.

Some other measures that have been introduced as size measures do not satisfy the above properties. Instances are the Estimator of length, and Volume [H77], which are not additive when software modules are disjoint (property Size.3). Indeed, for both measures, the value obtained when two disjoint software modules are concatenated may be less than the sum of the values obtained for each module, since they may contain common operators or operands. Note that, in this context, the graph is just the sequence of operand and operator occurrences. Disjoint code segments are disjoint subgraphs.

On the other hand, other measures, that are meant to capture other concepts, are indeed size measures. For instance, in the object-oriented suite of measures defined in [CK94], Weighted Methods per Class (WMC) is defined as the sum of the complexities of methods in a class. Implicitly, the program is seen as a directed acyclic graph (a hierarchy) whose terminal nodes are methods, and whose nonterminal nodes are classes. When two classes without methods in common are merged, the resulting class's WMC is equal to the sum of the two WMC's of the original classes (property Size.3 is satisfied). When two classes with methods in common are merged, then the WMC of the resulting class may be lower than the sum of the WMC's of the two original classes (formula Size.VI). Therefore, since all size properties hold (it is straightforward to show that properties Size.1 and Size.2 are true for WMC), this is a class size measure. However, WMC does not satisfy our properties for complexity measures (see Section 3.3). Likewise, NOC (Number Of Children of a class) and Response For a Class (RFC) [CK94] are other size measures, according to our properties.

3.2. Length

Motivation

Properties Size.1 - Size.3 characterize the concept of size as is commonly intended in software engineering. Actually, the concept of size may have different interpretations in everyday life, depending on the measurement goal. For instance, suppose we want to park a car in a parallel parking space. Then, the "size" we are interested in is the maximum distance between two points of the car linked by a segment parallel to the car's motion direction. The above properties Size.1 - Size.3 do not aim at defining such a measure of size. With respect to physical objects, volume and weight satisfy the above properties. In the particular case that the objects are unidimensional (or that we are interested in carrying out measurements with respect to only one dimension), then these concepts coincide.

In order to differentiate this measurement concept from size, we call it *length*. Length is non-negative (property Length.1), and equal to 0 when there are no elements in the system (property Length.2). In extreme situations where systems are composed of unrelated elements this property allows length to be non-null. If a new relationship is introduced between two elements belonging to the same connected component[2] of the graph representing a system, the length of the new system is not greater than the length of the original system (property Length.3). The idea is that, in this case, a new relationship may make the elements it connects "closer" than they were. This new relationship may reduce the maximum distance between elements in the connected component of the graph, but it may never increase it. On the other hand, if a new relationship is introduced between two elements belonging to two different connected components, the length of the new system is not smaller than the length of the original system. This stems from the fact that the new relationship creates a new connected component, where the maximum distance between two elements cannot be less than the maximum distance between any two elements of either original connected component (property Length.4). Length is not additive for disjoint modules. The length of a system containing several disjoint modules is the maximum length among them (property Length.5).

Definition 4: Length
The length of a system S is a function Length(S) characterized by the following properties Length.1 - Length.4.

◊

Property *Length.1*: Non-negativity
The length of a system S = <E,R> is non-negative

$$\text{Length}(S) \geq 0 \qquad \text{(Length.I)}$$

◊

Property *Length.2*: Null Value
The length of a system S = <E,R> is null if E is empty

$$(E = \emptyset) \Rightarrow (\text{Length}(S) = 0) \qquad \text{(Length.II)}$$

◊

Property *Length.3*: Non-increasing Monotonicity for Connected Components
Let S be a system and m be a module of S such that m is represented by a connected component of the graph representing S. Adding relationships between elements of m does not increase the length of S

[2]Here, two elements of a system S are said to belong to the same connected component if there is a path from one to the other in the non-directed graph obtained from the graph representing S by removing directions in the arcs.

$(S = \langle E,R \rangle$ **and** $m = \langle E_m,R_m \rangle$ **and** $m \subseteq S$
and m "is a connected component of S" **and**
$S' = \langle E,R' \rangle$ **and** $R' = R \cup \{\langle e_1,e_2 \rangle\}$ **and** $\langle e_1,e_2 \rangle \notin R$
and $e_1 \in E_{m1}$ **and** $e_2 \in E_{m1}) \Rightarrow \text{Length}(S) \geq \text{Length}(S')$ (Length.III)

◊

Property *Length.4*: Non-decreasing Monotonicity for Non-connected Components
Let S be a system and m_1 and m_2 be two modules of S such that m_1 and m_2 are represented by two separate connected components of the graph representing S. Adding relationships from elements of m_1 to elements of m_2 does not decrease the length of S

$(S = \langle E,R \rangle$ **and** $m_1 = \langle E_{m1},R_{m1} \rangle$ **and** $m_2 = \langle E_{m2},R_{m2} \rangle$
and $m_1 \subseteq S$ **and** $m_2 \subseteq S$ "are separate connected components of S" **and**
$S' = \langle E,R' \rangle$ **and** $R' = R \cup \{\langle e_1,e_2 \rangle\}$ **and** $\langle e_1,e_2 \rangle \notin R$
and $e_1 \in E_{m1}$ **and** $e_2 \in E_{m2}) \Rightarrow \text{Length}(S') \geq \text{Length}(S)$ (Length.IV)

◊

Property *Length.5*: Disjoint Modules
The length of a system S = <E,R> made of two disjoint modules m_1, m_2 is equal to the maximum of the lengths of m_1 and m_2

$(S = m_1 \cup m_2$ **and** $m_1 \cap m_2 = \emptyset$ **and** $E = E_{m1} \cup E_{m2}) \Rightarrow$
$\text{Length}(S) = \max\{\text{Length}(m_1),\text{Length}(m_2)\}$ (Length.V)

◊

Let us illustrate the last three properties with systems S, S', S", represented in Figure 3. We will assume that $m_1 = m'_1 = m''_1$, $m_2 = m'_2 = m''_2$, and $m_3 = m'_3 = m''_3$. The length of system S, composed of the three connected components m_1, m_2, and m_3, is the maximum value among the lengths of m_1, m_2, and m_3 (property Length.V). System S' differs from system S only because of the added relationship <c,m> (represented by the thick dashed arrow), which connects two elements already belonging to a connected component of S, m_3. The length of system S' is not greater than the length of S (property Length.III). System S" differs from system S only because of the added relationship <b,f> (represented by the thick solid arrow), which connects two elements belonging to two different connected components of S, m_1 and m_2. The length of system S" is not less than the length of S (property Length.IV).

Properties Length.1 - Length.5 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of length and the definition of length measures on a ratio scale.

Examples of length measures

Several measures can be defined at the system or module level based on the length concept. A typical example is the depth of a hierarchy. Therefore, the nesting depth in a program [F91] and DIT (Depth of Inheritance Tree—which is actually a hierarchy, in the general case) defined in [CK94] are length measures.

3.3. *Complexity*

Motivation

Intuitively, complexity is a measurement concept that is considered extremely relevant to system properties. It has been studied by several researchers (see Section 4 for a comparison between our framework and the literature). In our framework, we expect complexity to be non-negative (property Complexity.1) and to be null (property Complexity.2) when there are no relationships between the elements of a system. However, it could be argued that the complexity of a system whose elements are not connected to each other does not need to be necessarily null, because each element of E may have some complexity of its own. In our view, complexity is a system property that depends on the relationships between elements, and is not an isolated element's property. The complexity that an element taken in isolation may—intuitively—bring can only originate from the relationships between its "subelements." For instance, in a modular system, each module can be viewed as a "high-level element" encapsulating "subelements." However, if we want to consider the system as composed of such "high-level elements" (E), we should not "unpack" them, but only consider them and their relationships, without considering their "subelements" (E'). Otherwise, if we want to consider the contribution of the relationships between "subelements" (R'), we actually have to represent the system as S = <E', R∪R'>.

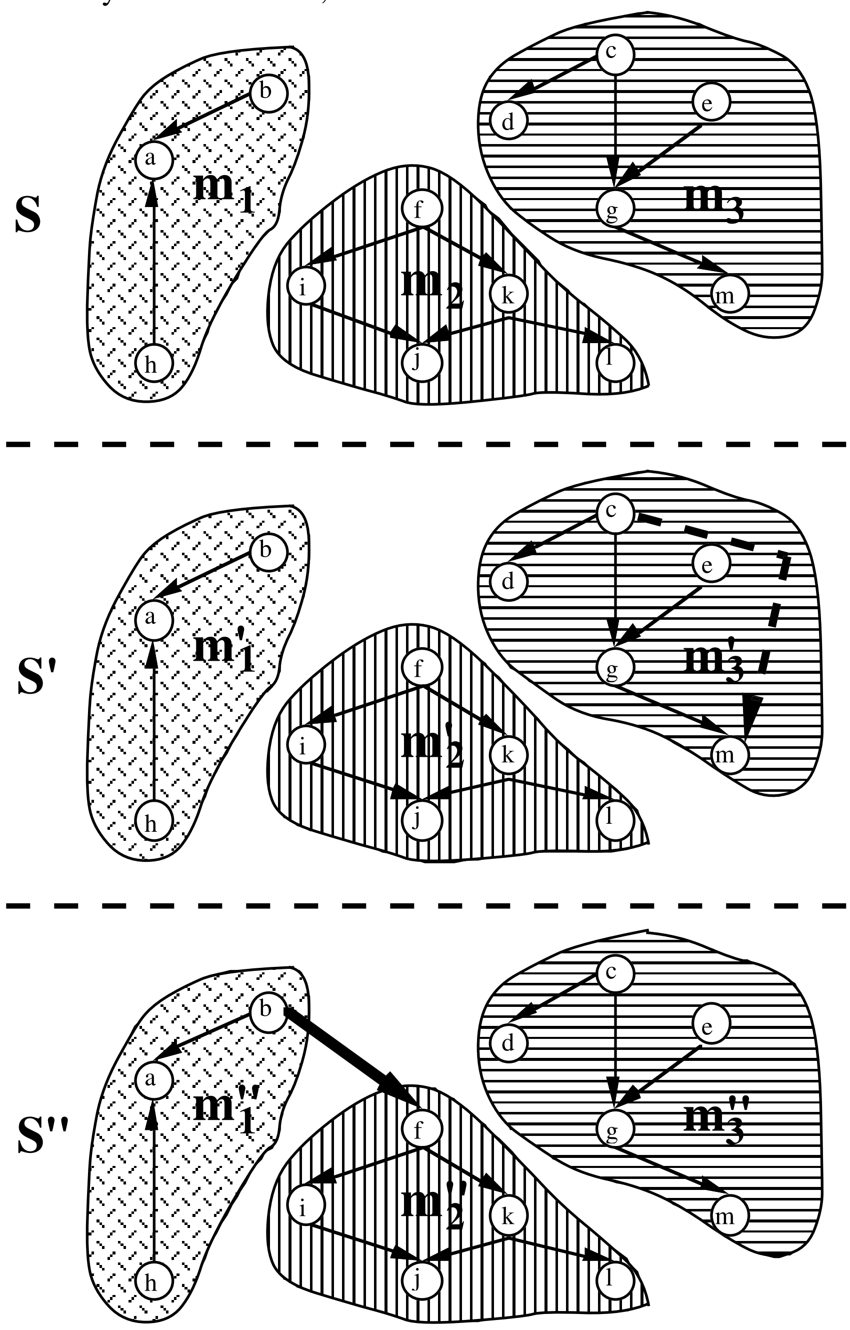

Figure 3. Properties of length.

Complexity should not be sensitive to representation conventions with respect to the direction of arcs representing system relationships (property Complexity.3). A relation can be represented in either an "active" (R) or "passive" (R^{-1}) form. The system and the relationships between its elements are not affected by these two equivalent representation conventions, so a complexity measure should be insensitive to this.

Also, the complexity of a system S should be at least as much as the sum of the complexities of any collections of its modules, such that no two modules share relationships, but may only share elements (property Complexity.4). *We believe that this property is the one that most strongly differentiates complexity from the other system concepts.* Intuitively, this property may be explained by two phenomena. First, the transitive closure of R is a larger graph than the graph obtained as the union of the transitive closures of R' and R'' (where R' and R'' are contained in R). As a consequence, if any kind of *indirect* (i.e., *transitive*) relationships between elements is considered in the computation of complexity, then the complexity of S may be larger than the sum of its modules' complexities**,** when the modules do not share any relationship. Otherwise, they are equal. Second, merging modules may implicitely generate between the elements of each modules. (e.g., definition-use relationships may be created when blocks are merged into a common system). As a consequence of the above properties, system complexity should not decrease when the set of system relationships is increased (property Complexity.4).

However**,** it has been argued that it is not always the case that the more relationships between the elements of a system, the more complex the system. For instance, it has been argued that adding a relationship between two elements may make the understanding of the system easier, since it clarifies the relationship between the two. This is certainly true, but we want to point out that this assertion is related to understandability, rather than complexity, and that complexity is only *one* of the factors that contribute to understandability. There are other factors that have a strong influence on understandability, such as the amount of available context information and knowledge about a system. In the literature [MGB90], it has been argued that the inner loop of the ShellSort algorithm, taken in isolation, is less understandable than the whole algorithm, since the role of the inner loop in the algorithm cannot be fully understood without the rest of the algorithm. This shows that understandability improves because a larger amount of context information is available, rather than because the complexity of the ShellSort algorithm is less than that of its inner loop. As another example, a relationship between two elements of a system may be added to *explicitly* state a relationship between them that was implicit or uncertain. This adds to our knowledge of the system, while, at the same time, increases complexity (according to our properties). In some cases (see above examples), the gain in context information/knowledge may overcome the increase in complexity and, as a result, may improve understandability. This stems from the fact that several phenomena concurrently affect understandability and does not mean in any way that an increase in complexity increases understandability.

Last, the complexity of a system made of disjoint modules is the sum of the complexities of the single modules (property Complexity.5). Consistent with property Complexity.4, this property is intuitively justified by the fact that the transitive closure of a graph composed of several disjoint subgraphs is equal to the union of the transitive closures of each subgraph taken in isolation. Furthermore, if two modules are put together in the same system, but they are not merged, i.e., they are still two disjoint module in this system, then no additional relationships are generated from the elements of one to the elements of the other.

The properties we define for complexity are, to a limited extent**,** a generalization of the properties several authors have already provided in the literature (see [LJS91, TZ92, W88]) for software code complexity, usually for control flow graphs. We generalize them because we may want to use them on artifacts other than software code and on abstractions other than control flow graphs.

Definition 5: Complexity
The complexity of a system S is a function Complexity(S) that is characterized by the following properties Complexity.1 - Complexity.5.

◊

Property *Complexity*.1: Non-negativity
The complexity of a system S = <E,R> is non-negative

$$\text{Complexity}(S) \geq 0 \qquad \text{(Complexity.I)}$$

◊

Property *Complexity*.2: Null Value
The complexity of a system S = <E,R> is null if R is empty

$$R = \emptyset \Rightarrow \text{Complexity}(S) = 0 \qquad \text{(Complexity.II)}$$

◊

Property *Complexity*.3: Symmetry
The complexity of a system S = <E,R> does not depend on the convention chosen to represent the relationships between its elements

$$(S=<E,R> \textbf{ and } S^{-1}=<E,R^{-1}>) \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1}) \qquad \text{(Complexity.III)}$$

◊

Property *Complexity*.4: Module Monotonicity
The complexity of a system S = <E,R> is no less than the sum of the complexities of any two of its modules with no relationships in common

$$\begin{aligned}&(S = <E,R> \textbf{ and } m_1 = <E_{m1},R_{m1}> \textbf{ and } m_2 = <E_{m2},R_{m2}> \\ &\quad \textbf{and } m_1 \cup m_2 \subseteq S \textbf{ and } R_{m1} \cap R_{m2} = \emptyset) \\ &\qquad \Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1)+\text{Complexity}(m_2) \qquad \text{(Complexity.IV)}\end{aligned}$$

◊

For instance, the complexity of the system shown in Figure 4 is not smaller than the sum of the complexities of m_1 and m_2.

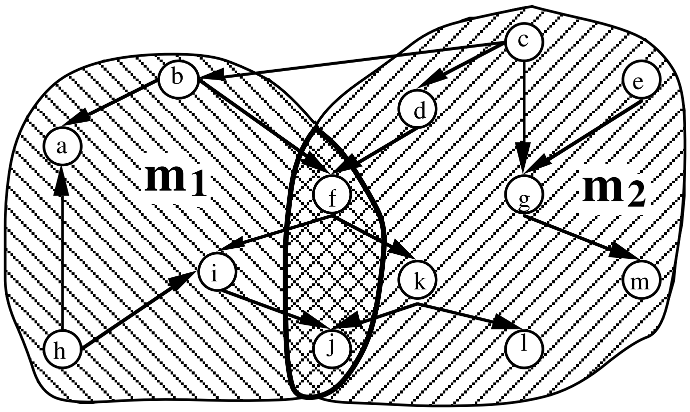

Figure 4. Module monotonicity of complexity.

Property *Complexity*.5: Disjoint Module Additivity
The complexity of a system S = <E,R> composed of two disjoint modules m_1, m_2 is equal to the sum of the complexities of the two modules

$$\begin{aligned}&(S = <E,R> \textbf{ and } S = m_1 \cup m_2 \textbf{ and } m_1 \cap m_2 = \emptyset) \\ &\quad \Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2) \qquad \text{(Complexity.V)}\end{aligned}$$

◊

For instance, the complexity of system S in Figure 2 is the sum of the complexities of its modules m_1, m_2, and m_3.

As a consequence of the above properties Complexity.1 - Complexity.5, it can be shown that adding relationships between elements of a system does not decrease its complexity

$$(S' = \langle E,R' \rangle \textbf{ and } S'' = \langle E,R'' \rangle \textbf{ and } R' \subseteq R'') \Rightarrow \text{Complexity}(S') \leq \text{Complexity}(S'') \qquad (\text{Complexity.VI})$$

Properties Complexity.1 - Complexity.5 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of complexity and the definition of complexity measures on a ratio scale.

Comprehensive comparisons and discussions of previous work in the area of complexity properties are provided in Section 4.

Examples and counterexamples of complexity measures

In [O80], Oviedo proposed a data flow complexity measure (DF). In this case, systems are programs, modules are program blocks, elements are variable definitions or uses, and relationships are defined between the definition of a given variable and its uses. The measure in [O80] is simply defined as the number of definition-use pairs in a block or a program. Property Complexity.4 holds. Given two modules (i.e., program blocks) which may only have common elements (i.e., no definition-use relationship is contained in both), the whole system (i.e., program) has a number of relationships (i.e., definition-use relationships) which is at least equal to the sum of the numbers of definition-use relationships of each module. Property Complexity.5 holds as well. The number of definition-use relationships of a system composed of two disjoint modules (i.e., blocks between which no definition-use relationship exists), is equal to the sum of the numbers of definition-use relationships of each module. As a conclusion, DF is a complexity measure according to our definition.

In [McC76], McCabe proposed a control flow complexity measure. Given a control flow graph $G = \langle E,R \rangle$ (which corresponds—unchanged—to a system for our framework), Cyclomatic Complexity is defined as

$$v(G) = |R| - |E| + 2p$$

where p is the number of connected components of G. Let us now check whether v(G) is a complexity measure according to our definition. It is straightforward to show that, except Complexity.4, the other properties hold. In order to check property Complexity.4, let $G = \langle E,R \rangle$ be a control flow graph and $G_1 = \langle E_1,R_1 \rangle$ and $G_2 = \langle E_2,R_2 \rangle$ two non-disjoint control flow subgraphs of G such that they have nodes in common but no relationships. We have to require that G_1 and G_2 be control flow subgraphs, because cyclomatic complexity is defined only for control flow graphs, i.e., graphs composed of connected components, each of which has a start node—a node with no incoming arcs—and an end node—a node with no outgoing arcs. Property Complexity.4 requires that the following inequality be true for all such G_1 and G_2

$$|R| - |E| + 2p \geq |R_1| - |E_1| + 2p_1 + |R_2| - |E_2| + 2p_2$$

i.e., $2(p_1 + p_2 - p) \leq |E_1| + |E_2| - |E|$, where p_1 and p_2 are the number of connected components in G_1 and G_2, respectively. This is not always true. For instance, consider Figure 5. G has 3 elements and 1 connected component; G_1 and G_2 have 2 nodes and 1 connected component apiece. Therefore, the above inequality is not true in this case, and the cyclomatic number is not a complexity measure according to our definition. However, it can be shown that v(G)-p satisfies all the above complexity properties. From a practical perspective, especially in large systems, this correction does not have a significant impact on the value of the measure.

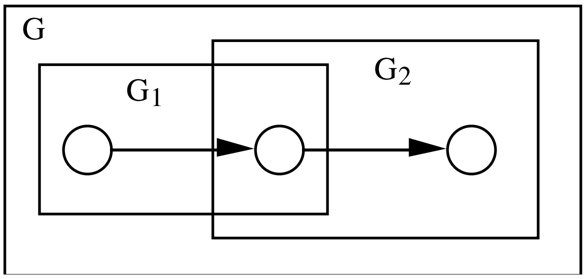

Figure 5. Control flow graph.

Henry and Kafura [HK81] proposed an information flow complexity measure. In this context, elements are subprogram variables or parameters, modules are subprograms, relationships are either fan-in's or fan-out's. For a subprogram SP, the complexity is expressed as *length.(fan-in.fan-out)*^{2}, where fan-in and fan-out are, respectively, the local (as defined in [HK81]) information flows from other subprograms to SP, and from SP to other subprograms. Such local information flows can be represented as relationships between parameters/variables of SP and parameters/variables of the other subprograms. Subprograms' parameters/variables are the system elements and the subprograms' fan-in and fan-out links are the relationships. Any size measure can be used for *length* (in [HK81] LOC was used). The justification for multiplying *length* and *(fan-in.fan-out)*^{2} was that "The complexity of a procedure depends on two factors: the complexity of the procedure code and the complexity of the procedure's connections to its environment." The complexity of the procedure code is taken into account by *length*; the complexity of the subprogram's connections to its environment is taken into account by *(fan-in.fan-out)*^{2}. The complexity of a system is defined as the sum of the complexities of the individual subprograms. For the measure defined above, properties Complexity.1 - Complexity.4 hold. However, property Complexity.5 does not hold since, given two disjoint modules S_1 and S_2 with a measured information flow of, respectively, $length_1.(fan\text{-}in_1.fan\text{-}out_1)^2$ and $length_2.(fan\text{-}in_2.fan\text{-}out_2)^2$, the following statement is true:

$$\text{length.(fan-in.fan-out)}^2 \geq \text{length}_1.(\text{fan-in}_1.\text{fan-out}_1)^2 + \text{length}_2.(\text{fan-in}_2.\text{fan-out}_2)^2$$

where length = $\text{length}_1 + \text{length}_2$, fan-in = $\text{fan-in}_1 + \text{fan-in}_2$, and fan-out = $\text{fan-out}_1 + \text{fan-out}_2$.

However, equality does not hold because of the exponent 2, which is not fully justified, and multiplication of fan-in and fan-out. Therefore, Henry and Kafura [HK81] information flow measure is not a complexity measure according to our definition. However, fan-in and fan-out taken as separate measures, without exponent 2, are complexity measures according to our definition since all the required properties hold.

Similar measures have been used in [C90] and referred to as *structural complexity* (SC) and defined as:

$$SC = \frac{\sum_{i \in [1..n]} \text{fan-out}^2(\text{subroutine}_i)}{n}$$

Once again, property Complexity.5 does not hold because fan-out is squared in the formula.

A metric suite for object-oriented design is proposed in [CK94]. A system is an object oriented design, modules are classes, elements are either methods or instance variables (depending on the measure considered) and relationships are calls to methods or uses of instance variables by other methods. These measures are validated against Weyuker's properties for complexity measures, thereby implicitely implying that they were complexity measures. However, none of the measures defined by [CK94] is a complexity measure according to our properties:

- Weighted Methods per Class (WMC) and Number Of Children of a class (NOC) are size measures (see Section 3.1);
- Depth of Inheritance Tree (DIT) is a length measure (see Section 3.2);
- Coupling Between Object classes (CBO) is a coupling measure (see Section 3.4);
- Response For a Class (RFC) is a size and coupling measure (see Sections 3.1 and 3.5);
- Lack of COhesion in Methods (LCOM) cannot be classified in our framework. This is consistent with what was said in the introduction: our framework does not cover all possible measurement concepts.

This is not surprising. In [CK94], it is shown that all of the above measures do not satisfy Weyuker's property 9, which is a weaker form of property Complexity.4 (see Section 4).

3.4. Cohesion

Motivation

The concept of cohesion has been used with reference to modules or modular systems. It assesses the tightness with which "related" program features are "grouped together" in systems or modules. It is assumed that the better the programmer is able to encapsulate related program features together, the more reliable and maintainable the system [F91]. This assumption seems to be supported by experimental results [BMB94(a)]. Intuitively, we expect cohesion to be non-negative and, more importantly, to be normalized (property Cohesion.1) so that the measure is independent of the size of the modular system or module. Moreover, if there are no internal relationships in a module or in all the modules in a system, we expect cohesion to be null (property Cohesion.2) for that module or for the system, since, as far as we know, there is no relationship between the elements and there is no evidence they should be encapsulated together. Additional internal relationships in modules cannot decrease cohesion since they are supposed to be additional evidence to encapsulate system elements together (property Cohesion.3). When two (or more) modules showing no relationships between them are merged, cohesion cannot increase because seemingly unrelated elements are encapsulated together (property Cohesion.4).

Since the cohesion (and, as we will see in Section 3.5, the coupling) of modules and entire modular systems have similar sets of properties, both will be described at the same time by using brackets and the alternation symbol '|'. For instance, the notation [A|B], where A and B are phrases, will denote the fact that phrase A applies to module cohesion, and phrase B applies to entire system cohesion.

Definition 6: Cohesion of a [Module | Modular System]
The cohesion of a [module $m = \langle E_m, R_m \rangle$ of a modular system MS | modular system MS] is a function [Cohesion(m)|Cohesion(MS)] characterized by the following properties Cohesion.1-Cohesion.4.

◊

Property *Cohesion.1*: Non-negativity and Normalization
The cohesion of a [module $m = \langle E_m, R_m \rangle$ of a modular system $MS = \langle E,R,M \rangle$ | modular system $MS = \langle E,R,M \rangle$] belongs to a specified interval

$$[\ \mathrm{Cohesion}(m) \in [0,Max] \quad | \quad \mathrm{Cohesion}(MS) \in [0,Max]\ ] \qquad \text{(Cohesion.I)}$$

◊

Normalization allows meaningful comparisons between the cohesions of different [modules|modular systems], since they all belong to the same interval.

Property *Cohesion.2:* Null Value.
The cohesion of a [module m = $<E_m,R_m>$ of a modular system MS = <E,R,M> | modular system MS = <E,R,M>] is null if [R_m|IR] is empty

$$[\ R_m = \emptyset \Rightarrow \text{Cohesion}(m) = 0 \mid IR = \emptyset \Rightarrow \text{Cohesion}(MS) = 0\ ] \qquad \text{(Cohesion.II)}$$

(Recall that IR is the set of intra-module relationships, defined in Definition 2.)

◊

If there is no intra-module relationship among the elements of a (all) module(s), then the module (system) cohesion is null.

Property *Cohesion.3:* Monotonicity.
Let MS' = <E,R',M'> and MS" = <E,R",M"> be two modular systems (with the same set of elements E) such that there exist two modules m' = $<E_m,R_{m'}>$ and m" = $<E_m,R_{m''}>$ (with the same set of elements E_m) belonging to M' and M" respectively, such that R' - $R_{m'}$ = R" - $R_{m''}$, and $R_{m'} \subseteq R_{m''}$ (which implies IR' $\subseteq$ IR"). Then,

$$[\ \text{Cohesion}(m')\leq\text{Cohesion}(m'') \mid \text{Cohesion}(MS')\leq\text{Cohesion}(MS'')\ ] \qquad \text{(Cohesion.III)}$$

◊

Adding intra-module relationships does not decrease [module|modular system] cohesion. For instance, suppose that systems S, S', and S" in Figure 3 are viewed as modular systems MS = <E,R,M>, MS' = <E',R',M'>, and MS" = <E",R",M"> (with M = $\{m_1,m_2,m_3\}$, M' = $\{m'_1,m'_2,m'_3\}$, and M" = $\{m''_1,m''_2,m''_3\}$). We have [Cohesion(m'_3) ≥ Cohesion(m_3) | Cohesion(MS') ≥ Cohesion(MS)].

Property* Cohesion.4*: Cohesive Modules.
Let MS' = <E,R,M'> and MS" = <E,R,M"> be two modular systems (with the same underlying system <E,R>) such that M" = M' - $\{m'_1,m'_2\} \cup \{m''\}$, with $m'_1 \in$ M', $m'_2 \in$ M', m" $\notin$ M', and m" = $m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m", union of m'_1 and m'_2.) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., InputR(m'_1) $\cap$ OutputR(m'_2) = $\emptyset$ **and** InputR(m'_2) $\cap$ OutputR(m'_1) = $\emptyset$, then

$$[\ \max\{\text{Cohesion}(m'_1),\text{Cohesion}(m'_2)\} \geq \text{Cohesion}(m'') \mid \text{Cohesion}(MS') \geq \text{Cohesion}(MS'')\ ] \qquad \text{(Cohesion.IV)}$$

◊

The cohesion of a [module|modular system] obtained by putting together two unrelated modules is not greater than the [maximum cohesion of the two original modules|the cohesion of the original modular system].
Properties Cohesion.1 - Cohesion.4 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of cohesion and the definition of cohesion measures on a ratio scale.

Examples of cohesion measures

In [BMB94(a)], cohesion measures for high-level design are defined and validated, at both the abstract data type (module) and system (program) levels. For brevity's sake, the term software part here denotes either a module or a program. A high-level design is seen as a collection of modules, each of which exports and imports constants, types, variables, and procedures/functions. A widely accepted software engineering principle prescribes that each module be highly cohesive, i.e., its elements be tightly related to each other. [BMB94(a)] focuses on investigating whether high cohesion values are related to lower error-proneness, due to the fact that the changes required by a change in a module are confined in a well-encapsulated part of the overall program. To this end, the exported feature A is said to interact with feature B if the change of one of A's definitions or uses may require a change in one of B's definitions or uses.

In the approach of the present paper, each feature exported by a module is an element of the system, and the interactions between them are the relationships between elements. A module according to [BMB94(a)] is represented by a module according to the definition of the present paper. At high-level design time, not all interactions between the features of a module are known, since the features may interact in the body of a module, and not in its visible part. Given a software part sp, three cohesion measures NRCI(sp), PRCI(sp), and ORCI(sp) (respectively, Neutral, Pessimistic, and Optimistic Ratio of Cohesive Interactions) are defined for software as follows

$$\mathrm{NRCI(sp)} = \frac{\mathrm{\#KnownInteractions(sp)}}{\mathrm{\#MaxInteractions(sp)\text{-}\#UnknownInteractions(sp)}}$$

$$\mathrm{PRCI(sp)} = \frac{\mathrm{\#KnownInteractions(sp)}}{\mathrm{\#MaxInteractions(sp)}}$$

$$\mathrm{ORCI(sp)} = \frac{\mathrm{\#KnownInteractions(sp)+\#UnknownInteractions(sp)}}{\mathrm{\#MaxInteractions(sp)}}$$

where #MaxInteractions(sp) is the maximum number of possible intra-module interactions between the features exported by each module of the software part sp. (Inter-module interactions are not considered cohesive; they may contribute to coupling, instead.) All three measures satisfy the above properties Cohesion.1 - Cohesion.4.

Other examples of cohesion measures can be found in [BO94], where new functional cohesion measures are introduced. Given a procedure, function, or main program, only *data tokens* (i.e., the occurrence of a definition or use of a variable or a constant) are taken into account. The *data slice* for a data token is the sequence of all those data tokens in the program that can influence the statement in which the data token appears, or can be influenced by that statement. Being a sequence, a data slice is ordered: it lists its data tokens in order of appearance in the procedure, function or main program. If more than one data slice exists, some data tokens may belong to more than one data slice: these are called *glue tokens*. A subset of the glue tokens may belong to all data slices: these are called *super-glue tokens*. Functional cohesion measures are defined based on data tokens, glue tokens, and super-glue tokens. This approach can be represented in our framework as follows. A data token is an element of the system, and a data slice is represented as a sequence of nodes and arcs. The resulting graph is a Directed Acyclic Graph, which represents a module. ([BO94] introduces functional cohesion measures for single procedures, functions, or main programs.) Given a procedure, function, or main program p, the following measures SFC(p) (Strong Functional Cohesion), WFC(p) (Weak Functional Cohesion), and A(p) (adhesiveness) are introduced

$$\mathrm{SFC(p)} = \frac{\mathrm{\#SuperGlueTokens}}{\mathrm{\#AllTokens}}$$

$$\mathrm{WFC(p)} = \frac{\mathrm{\#GlueTokens}}{\mathrm{\#AllTokens}}$$

$$\mathrm{A(p)} = \frac{\sum_{\mathrm{GT} \in \mathrm{GlueTokens}} \mathrm{\#SlicesContainingGlueTokenGT}}{\mathrm{\#AllTokens.\#DataSlices}}$$

It can be shown that these measures satisfy the above properties Cohesion.1 - Cohesion.4.

3.5. Coupling

Motivation

The concept of coupling has been used with reference to modules or modular systems. Intuitively, it captures the amount of relationship between the elements belonging to different modules of a system. Given a module m, two kinds of coupling can be defined: inbound coupling and outbound coupling. The former captures the amount of relationships from elements outside m to elements inside m; the latter the amount of relationships from elements inside m to elements outside m.

We expect coupling to be non-negative (property Coupling.1), and null when there are no relationships among modules (property Coupling.2). When additional relationships are created across modules, we expect coupling not to decrease since these modules become more interdependent (property Coupling.3). Merging modules can only decrease coupling since there may exist relationships among them and therefore, inter-module relationships may have disappeared (property Coupling.4, property Coupling.5).

In what follows, when referring to module coupling, we will use the word coupling to denote either inbound or outbound coupling, and OuterR(m) to denote either InputR(m) or OutputR(m).

Definition 7: Coupling of a [Module | Modular System]
The coupling of a [module m = $<E_m,R_m>$ of a modular system MS|modular system MS] is a function [Coupling(m)|Coupling(MS)] characterized by the following properties Coupling.1 - Coupling.5.

◊

Property *Coupling.1*: Non-negativity
The coupling of a [module m = $<E_m,R_m>$ of a modular system|modular system MS] is non-negative

[Coupling(m) ≥ 0 | Coupling(MS) ≥ 0] (Coupling.I)

◊

Property *Coupling.2*: Null Value
The coupling of a [module m = $<E_m,R_m>$ of a modular system|modular system MS = $<E,R,M>$] is null if [OuterR(m)|R-IR] is empty

[OuterR(m)=∅ ⇒ Coupling(m)=0 | R-IR=∅ ⇒ Coupling(MS)=0] (Coupling.II)

◊

Property *Coupling.3*: Monotonicity
Let MS' = $<E,R',M'>$ and MS" = $<E,R'',M''>$ be two modular systems (with the same set of elements E) such that there exist two modules m' ∈ M', m" ∈ M" such that R' - OuterR(m') = R" - OuterR(m"), and OuterR(m') ⊆ OuterR(m"). Then,

[Coupling(m')≤Coupling(m") | Coupling(MS')≤Coupling(MS")] (Coupling.III)

◊

Adding inter-module relationships does not decrease coupling. For instance, if systems S, and S" in Figure 3 are viewed as modular systems (see Section 3.4), we have [Coupling(m''_1) ≥ Coupling(m_1) | Cohesion(MS") ≥ Cohesion(MS)].

Property *Coupling.4*: Merging of Modules
Let MS' = $<E',R',M'>$ and MS" = $<E'',R'',M''>$ be two modular systems such that E' = E", R' = R", and M" = M' - $\{m'_1,m'_2\} \cup \{m''\}$, where $m'_1 = <E_{m'1},R_{m'1}>$, $m'_2 = <E_{m'2},R_{m'2}>$, and $m'' = <E_{m''},R_{m''}>$, with $m'_1 \in M'$, $m'_2 \in M'$, $m'' \notin M'$, and $E_{m''} = E_{m'1} \cup E_{m'2}$ and $R_{m''} = R_{m'1} \cup$

$R_{m'2}$. (The two modules m'_1 and m'_2 are replaced by the module m", whose elements and relationships are the union of those of m'_1 and m'_2.) Then

[$Coupling(m'_1) + Coupling(m'_2) \geq Coupling(m")$ |
$Coupling(MS') \geq Coupling(MS")$] (Coupling.IV)

◊

The coupling of a [module|modular system] obtained by merging two modules is not greater than the [sum of the couplings of the two original modules|coupling of the original modular system], since the two modules may have common inter-module relationships. For instance, suppose that the modular system MS_{12} in Figure 6 is obtained from the modular system MS in Figure 2 by merging modules m_1 and m_2 into module m_{12}. Then, we have [$Coupling(m_1) + Coupling(m_2) \geq Coupling(m_{12})$ | $Coupling(MS) \geq Coupling(MS_{12})$].

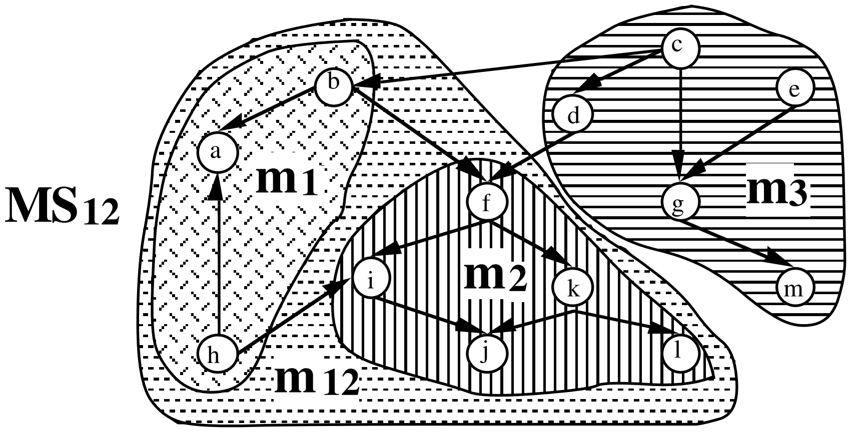

Figure 6. The effect of merging modules on coupling.

Property *Coupling.5*: Disjoint Module Additivity
Let MS' = <E,R,M'> and MS" = <E,R,M"> be two modular systems (with the same underlying system <E,R>) such that $M" = M' - \{m'_1, m'_2\} \cup \{m"\}$, with $m'_1 \in M'$, $m'_2 \in M'$, $m" \notin M'$, and $m" = m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m", union of m'_1 and m'_2.) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $InputR(m'_1) \cap OutputR(m'_2) = \emptyset$ **and** $InputR(m'_2) \cap OutputR(m'_1) = \emptyset$, then

[$Coupling(m'_1) + Coupling(m'_2) = Coupling(m")$ |
$Coupling(MS') = Coupling(MS")$] (Coupling.V)

◊

The coupling of a [module|modular system] obtained by merging two unrelated modules is equal to the [sum of the couplings of the two original modules|coupling of the original modular system].

Properties Coupling.1 - Coupling.5 hold when applying the admissible transformations of the ratio scale. Therefore, there is no contradiction between our concept of coupling and the definition of coupling measures on a ratio scale.

Examples and counterexamples of coupling measures

Fenton has defined an ordinal coupling measure between pairs of subroutines [F91] as follows:

$$C(S, S') = i + \frac{n}{n+1}$$

where i is the number corresponding to the worst coupling type (according to Myers' ordinal scale [F91]) and n the number of interconnections between S and S', i.e., global variables and formal parameters. In this case, systems are programs, modules are subroutines, elements are formal parameters and global variables. If coupling for the whole system is defined as the sum of coupling values between all subroutine pairs, properties Coupling.1 - Coupling.5 hold for this measures and we label it as a coupling measure. However, Fenton proposes to calculate the median of all the pair values as a system coupling measure. In this case, property Coupling.3 does not hold since the median may decrease when inter-module relationships are added. Similarly for Coupling.4, when subroutines are merged and inter-module relationships are lost, the median may increase. Therefore, the system coupling measure proposed by Fenton is not a coupling measure according to our definitions.

In [BMB94(a)], coupling measures for high-level design are defined and validated, at both the module (abstract data type) and system (program) levels. They are based on the notion of interaction introduced in the examples of Section 3.4. Import Coupling of a module m is defined as the extent to which m depends on imported external data declarations. Similarly, export coupling of m is defined as the extent to which m's data declarations affect the other data declarations in the system. At the system level, coupling is the extent to which the modules are related to each other. Given a module m, Import Coupling of m (denoted by IC(m)) is the number of interactions between data declarations external to m and the data declarations within m. Given a module m, Export Coupling of m (denoted by EC(m)) is the number of interactions between the data declarations within m and the data declarations external to m. As shown in [BMB94(a)], our coupling properties hold for these measures.

Coupling Between Object classes (CBO) of a class is defined in [CK94] as the number of other classes to which it is coupled. It is a coupling measure. Properties Coupling.1 and Coupling.2 are obviously satisfied. Property Coupling.3 is satisfied, since CBO cannot decrease by adding one more relationship between features belonging to different classes (i.e., one class uses one more method or instance variable belonging to another class). Property Coupling.4 is satisfied: CBO can only remain constant or decrease when two classes are grouped into one. Property Coupling.4 is also satisfied.

Response For a Class (RFC) [CK94] is a size and a coupling measure at the same time (see Section 3.1). Methods are elements, calls are relationships, classes are modules. Coupling.3 holds, since adding outside method calls to a class can only increase RFC and Coupling.4 holds because merging classes does not change RFC's value since RFC does not distinguish between inside and outside method calls. Similarly, when there are no calls between the classes' methods, Coupling.5 holds. This result is to be expected since RFC is the result of the addition of two terms: the number of methods in the class, a size measure, and the number of methods called, a coupling measure.

3.6. Comparison of Concept Properties

We want to summarize the important differences and similarities between the system concepts introduced in this paper. Table 1 uses only criteria that can be compared across the concepts of size, length, complexity, cohesion, and coupling. First, it is important to recall that coupling and cohesion are only defined in the context of modular systems, whereas size, length and complexity are defined for all systems.

Second, the concepts appear to have the null value (second column) and monotonicity (third column) properties based on different sets. The behavior of a measure with respect to variations in such sets characterizes the nature of the measure itself, i.e., the concept(s) it captures. As RFC, defined in [CK94], shows (see Sections 3.1 and 3.5), the same measure may satisfy the sets of properties associated with different concepts. As a matter of fact, similar sets of properties associated with different concepts are not contradictory.

Third, when systems are made of disjoint modules, size, complexity and coupling are additive (properties Size.3, Complexity.5, and Coupling.5). Cohesion and length are not additive.

Concepts\Properties	Null Value	Monotonicity	Additivity
Size	$E = \emptyset$	E	Yes
Length	$E=\emptyset$	R	No
Complexity	$R = \emptyset$	R	Yes
System Cohesion	$IR=\emptyset$	IR	No
System Coupling	$R\text{-}IR=\emptyset$	R-IR	Yes

Table 1: Comparison of concept properties

This summary shows that these concepts are really different with respect to basic properties. Therefore, it appears that desirable properties are likely to vary from one measurement concept to another.

4. Comparison with Related Work

We mainly compare our approach with the other approaches for defining sets of properties for software complexity measures, because they have been studied more extensively and thoroughly than other kinds of measures. Besides, we compare our approach with the axioms introduced by Fenton and Melton [FM90] for software coupling measures. As already mentioned, our approach generalizes previous work on properties for defining complexity measures. Unlike previous approaches, it is not constrained to deal with software code only, but, because of its generality, can be applied to other artifacts produced during the software lifecycle, namely, software specifications and designs. Moreover, it is not defined based on some control flow operations, like sequencing or nesting, but on a general representation, i.e., a graph.

Weyuker[3]

Weyuker's work [W88] is one of the first attempts to formalize the fuzzy concept of program complexity. This work has been discussed by many authors [CK94, F91, LJS91, TZ92, Z91] and is still a point of reference and comparison for anyone investigating the topic of software complexity.

To make Weyuker's properties comparable with ours, we will assume that a program according to Weyuker is a system according to our definition; a program body is a module of a system. A whole program is built by combining program bodies, by means of sequential, conditional, and iterative constructs (plus the program and output statements, which can be seen as "special" program bodies), and, correspondingly, a system can be built from its constituent modules. Since some of Weyuker's properties are based on the sequencing between pairs of program bodies P and Q, we provide more details about the representation of sequencing in our framework. Sequencing of program bodies P and Q is obtained via the composition operation (P;Q). Correspondingly, if $S_P = \langle E_P,R_P \rangle$ and $S_Q = \langle E_Q,R_Q \rangle$ are the modules representing the two program bodies P and Q[4], then, we will denote the representation of P;Q as $S_{P;Q} = \langle E_{P;Q},R_{P;Q} \rangle$. In what follows, we will assume that $E_{P;Q} = E_P \cup E_Q$ and $R_{P;Q}$ $R_P \cup R_Q$, i.e., the representation of the composition of two program bodies contains the elements of the representation of each program body, and at least contains all the relationships belonging to each of the representations of program bodies. In other words, S_P and S_Q are modules of $S_{P;Q}$.

[3]We will list properties/axioms by the initial of the proponents. So, Weyuker's properties will be referred to as W1, W2, …, W9, Tian and Zelkowitz's as TZ1 to TZ5, and Lakshmanian et alii's as L1 to L9.

[4] In what follows, we will use the notation $S_P = \langle E_P,R_P \rangle$ to denote the representation of program body P.

W1: *A complexity measure must not be "too coarse" (1).*
$\exists\ S_P, S_Q$ Complexity(S_P) $\neq$ Complexity(S_Q)

W2: *A complexity measure must not be "too coarse" (2).* Given the nonnegative number c, there are only finitely many systems of complexity c.

W3: *A complexity measure must not be "too fine."* There are distinct systems S_P and S_Q such that Complexity(S_P) = Complexity(S_Q).

W4: *Functionality*. There is no one-to-one correspondence between functionality and complexity
$\exists\ S_P,S_Q$ P and Q are functionally equivalent **and** Complexity(S_P) $\neq$ Complexity(S_Q)

W5: *Monotonicity with respect to composition.*
$\forall\ S_P,S_Q$
Complexity(S_P) $\leq$ Complexity($S_{P;Q}$) **and** Complexity(S_Q) $\leq$ Complexity($S_{P;Q}$)

W6: *The contribution of a module in terms of the overall system complexity may depend on the rest of the system.*
(a) $\exists\ S_P, S_Q, S_T$ Complexity(S_P) = Complexity(S_Q) **and** Complexity($S_{P;T}$) $\neq$ Complexity($S_{Q;T}$)

(b) $\exists\ S_P, S_Q, S_T$ Complexity(S_P) = Complexity(S_Q) **and** Complexity($S_{T;P}$) $\neq$ Complexity($S_{T;Q}$)

W7: *A complexity measure is sensitive to the permutation of statements.*
$\exists\ S_P, S_Q$ Q is formed by permuting the order of statements of P **and** Complexity(S_P) $\neq$ Complexity(S_Q)

W8: *Renaming*. If P is a renaming of Q, then Complexity(S_P)=Complexity(S_Q).

W9: *Module monotonicity.*
$\exists\ S_P, S_Q$ Complexity(S_P) + Complexity(S_Q) $\leq$ Complexity($S_{P;Q}$)

Analysis of Weyuker's properties

W1, W2, W3, W4, W8: These are not implied by our properties, but they do not contradict any of them, so they can be added to our set, if desired. However, we think that these properties are general to all syntactically-based product measures and do not appear useful in our framework to differentiate concepts.

W5: This is implied by our properties, as shown by inequality (Complexity.VI), since S_P and S_Q are modules of $S_{P;Q}$.

W6, W7: These properties are not implied by the above properties Complexity.1 - Complexity.5. However, they show a very important and delicate point in the context of complexity measure definition.

By assuming properties W6(a) and W6(b) to be false, one forces all complexity measures to be strongly related to control flow, since this would exclude that the composition of two program bodies may yield additional relationships between elements (e.g., data declarations) of the two program bodies. If properties W6(a) and W6(b) are assumed true, one forces all complexity measures to be sensitive to at least one other kind of additional relationship.

Similarly, W7 states that the order of the statements, and therefore the control flow, should have an impact on all complexity measures. By assuming property W7 to be false, one forces all complexity measures to be insensitive to the ordering of statements. If property W7 is assumed true, one forces all complexity measures to be somehow sensitive to the ordering of statements, which may not always be useful.

W8: We analyze this property again, to better explain the relationship between complexity and understandability. According to this property, renaming does not affect complexity. However, it is a fact that renaming program variables by absurd or misleading names greatly impairs understandability. This shows that other factors, besides complexity, affect understandability and the other external qualities of software that are affected by complexity.

As for properties W1-W8, our approach is somewhat more liberal than Weyuker's. For instance, the constant null function is an acceptable complexity measure according to our properties, while it is not acceptable according to Weyuker's properties. It is evident that the usefulness of such a complexity measure is questionable. We think that properties should be used to check whether a measure actually addresses a given concept (e.g., complexity). However, given any set of properties, it is almost always possible to build a measure that satisfies them, but is of no practical interest (see [CS91]). At any rate, this is not a sensible reason to reject a set of properties associated with a concept (how many sensless measures could be defined that satisfy the three properties that characterize distance!). Rather, measures that satisfy a set of properties must be later assessed with regard to their usefulness.

W9: This is probably the most controversial property. The above properties Complexity.1 - Complexity.5 imply it. Actually, our properties imply the stronger form of W9, the unnumbered property following W9 in Weyuker's paper [W88] (see also [P84])

$\forall\ S_P, S_Q\ \mathrm{Complexity}(S_P) + \mathrm{Complexity}(S_Q) \leq \mathrm{Complexity}(S_{P;Q})$

Weyuker rejects it on the basis that it might lead to contradictions: she argues that the effort needed to implement or understand the composition of a program body P with itself, is probably not twice as much as the effort needed for P alone. Our point is that complexity is not the only factor to be taken into account when evaluating the effort needed to implement or understand a program, nor is it proven that this effort is in any way "proportional" to product complexity.

Fenton

In addition to Weyuker's work, Fenton [F94] shows that, based on measurement-theoretic mathematical grounds, there is no chance that a general measure for software complexity will ever be found, nor even for control flow complexity, i.e., a more specific kind of complexity. We totally agree with that. By no means do we aim at defining a single complexity measure, which captures all kinds of complexity in a software artifact. Instead, our set of properties define constraints for any specific complexity measure, whatever facet of complexity it addresses.

Fenton and Melton [FM90] introduced two axioms that they believe should hold for coupling measures. Both axioms assume that coupling is a measure of connectivity of a system represented by its module design chart (or structure chart). The first axiom is similar to our monotonicity property (Coupling.3). It states that if the only difference between two module design charts D and D' is an extra interconnection in D', then the coupling of D' is higher than the coupling of D. The second axiom basically states that system coupling should be independent from the number of modules in the system. If a module is added and shows the same level of pairwise coupling as the already existing modules, then the coupling of the system remains constant. According to our properties, coupling is seen as a measure which is to a certain extent dependent on the number of modules in the system and we therefore do not have any equivalent axiom. This shows that the sets of properties that can be defined above are, to some extent, subjective.

Zuse

In his article in the *Encyclopaedia of Software Engineering* [ESE94 pp. 131-165], Zuse applies a measurement-theoretic approach to complexity measures. The focus is on the conditions that should be satisfied by empirical relational systems in order to provide them with additive ratio scale measures. This class of measures is a subset of ratio scale measures, characterized by the additivity property (Theorems 2 and 3 of [ESE94]). Given the set P of flowgraphs and a binary operation * between flowgraphs (e.g., concatenation), additive ratio scale complexity measures are such that, for each pair of flowgraphs P1, P2,

Complexity(P1*P2) = Complexity(P1) + Complexity(P2)

This property shows that a different concept of complexity is defined by Zuse, with respect to that defined by Weyuker's (W9) and our properties (Complexity.4). It is our belief that, by requiring that complexity measures be additive, important aspects of complexity may not be fully captured, and complexity measures actually become quite similar to size measures. Considering complexity as additive means that, when two modules are put together to form a new system, no additional dependencies between the elements of the modules should be taken into account in the computation of the system complexity. We believe this is a very questionable assumption for product complexity.

Tian and Zelkowitz

Tian and Zelkowitz [TZ92] have provided axioms (necessary properties) for complexity measures and a classification scheme based on additional program characteristics that identify important measure categories. In the approach, programs are represented by means of their abstract syntax trees (e.g., parse trees). To translate this representation into our framework, we will assume that the whole program, represented by the entire tree, is a system, and that any part of a program represented by a subtree is a module.

TZ1: Systems with identical functionality are comparable, i.e., there is an order relation between them with respect to complexity.
TZ2: A system is comparable with its module(s).
TZ3: Given a system S_Q and any module S_P whose root, in the abstract tree representation, is “far enough” from the root of S_Q, then S_P is not more complex than S_Q. In other words, "small" modules of a system are no more complex than the system.
TZ4: If an intuitive complexity order relation exists between two systems, it must be preserved by the complexity measure (it is a weakened form of the representation condition of Measurement Theory [F91]).
TZ5: Measures must not be too coarse and must show sufficient variability.

TZ1, TZ2, TZ5 do not differentiate software characteristics (concepts) and can be used for all syntactic product measures. **TZ3** can be derived from our set of properties. **TZ4** captures the basic purpose behind the definition of all measures: preserving an intuitive order on a set of software artifacts [MGB90].

The additional set of properties which is presented in [TZ92] is used to define a measure classification system. It determines whether or not a measure is based exclusively on the abstract syntax tree of the program, whether it is sensitive to renaming, whether it is sensitive to the context of definition or use of the measured program, whether it is determined entirely by the performed program operations regardless of their order and organization, and whether concatenation of programs always contribute positively toward the composite program complexity (i.e., system monotonicity).

Some of these properties are related to the properties defined in this paper and we believe they are characteristic properties of distinct system concepts (e.g., system monotonicity). Others do not differentiate the various concepts associated with syntactically-based measures (e.g., renaming).

Lakshmanian et al.

Lakshmanian et al. [LJS91] have attempted to define necessary properties for software complexity measures based on control flow graphs. In order to make these properties comparable to ours, we will use a notation similar to the one used to introduce Weyuker's properties. A program according to Lakshmanian et al. (represented by a control flow graph) is a system according to our definition, and a program segment is a module. In addition to sequencing, these properties use the nesting program construct denoted as @. "A program segment Z is said to be obtained by nesting [program segment] Y at the control location i in [program segment] X (denoted by $Y@X_i$) if the program segment X has at least one conditional branch, and if Y is embedded at location i in X in such a way that there exists at least one control flow path in the combined code Z that completely skips Y." "The notation Y@X refers to any nesting of Y in X if the specific location in X at which Y is embedded is immaterial."

In what follows, X, Y, Z will denote programs or program segments; S_X, S_Y, S_Z will denote the corresponding systems or modules according to our definition. Lakshmanian et al. [LJS91] introduce nine properties. However, only five out of them can be considered basic, since the remaining four can be derived from them. Therefore, below we will only discuss the compatibility of the basic properties with respect to our properties.

L1: Non-negativity.

L1(a): Null value.
If the program only contains sequential code (referred to as a basic block B) then

$\text{Complexity}(S_B) = 0$

L1(b): Positivity.
If the program X is not a basic block, then

$\text{Complexity}(S_X) > 0$

◊

Property L1 does not contradict any of our properties (in particular, Complexity 1 and Complexity 2).

L5: Additivity under sequencing.
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_Y) + \text{Complexity}(S_X)$

◊

This property does not contradict properties Complexity.4 and Complexity.5, where the equality sign is allowed. By requiring that complexity be additive under sequencing, Lakshmanian et al take a viewpoint which is very similar to that of Zuse**.**

L6: Functional independence under nesting.
Adding a basic block B to a system X through nesting does not increase its complexity

$\text{Complexity}(S_{B@X}) = \text{Complexity}(S_X)$

◊

L7: Monotonicity under nesting.
$\text{Complexity}(S_{Y@X_i}) < \text{Complexity}(S_{Z@X_i})$ if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$

◊

These properties are compatible with our properties.

L9: Sensitivity to nesting.
$\text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{Y@X})$ if $\text{Complexity}(S_Y) > 0$

◊

This property does not contradict our properties.

In conclusion, none of the above properties contradicts our properties. However, the scope of these properties is limited to the sequencing and nesting of control flow graphs, and therefore to the study of control flow complexity.

As for the other properties, we now show how they can be derived from L1, L5, L6, L7, and L9.

L2: Functional independence under sequencing.
$\text{Complexity}(S_{X;B}) = \text{Complexity}(S_X)$

This property follows from L5 (first equality below) and L1 (second equality below):

$\text{Complexity}(S_{X;B}) = \text{Complexity}(S_X) + \text{Complexity}(S_B) = \text{Complexity}(S_X)$

◊

L3: Symmetry under sequencing.
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_{Y;X})$

This property follows from L5 (both equalities)
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y) = \text{Complexity}(S_{Y;X})$

◊

L4: Monotonicity under sequencing.
$\text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{X;Z})$ if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_{X;Z})$ if $\text{Complexity}(S_Y) = \text{Complexity}(S_Z)$

This property follows from L5:

if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$, then
$$\begin{aligned}\text{Complexity}(S_{X;Y}) &= \text{Complexity}(S_X) + \text{Complexity}(S_Y)\\ &< \text{Complexity}(S_X) + \text{Complexity}(S_Z) = \text{Complexity}(S_{X;Z})\end{aligned}$$
if $\text{Complexity}(S_Y) = \text{Complexity}(S_Z)$, then
$$\begin{aligned}\text{Complexity}(S_{X;Y}) &= \text{Complexity}(S_X) + \text{Complexity}(S_Y)\\ &= \text{Complexity}(S_X) + \text{Complexity}(S_Z) = \text{Complexity}(S_{X;Z})\end{aligned}$$

◊

L8: Monotonicity under nesting.
$\text{Complexity}(S_Y) < \text{Complexity}(S_{Y@X})$

This property follows from L1 (first inequality below, since $\text{Complexity}(S_X)>0$—X cannot be a basic block), L5 (equality below) and L9 (second inequality below)

$$\begin{aligned}\text{Complexity}(S_Y) &< \text{Complexity}(S_X) + \text{Complexity}(S_Y)\\ &= \text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{Y@X})\end{aligned}$$

◊

5. Conclusion and Directions for Future Work

In order to provide some guidelines for the analyst in charge of defining product measures, we propose a framework for software measurement where various software measurement concepts are distinguished and their specific properties defined in a generic manner. Such a framework is, by its very nature, somewhat subjective and there are possible alternatives to it. However, it is a practical framework since the properties we capture are, we believe, interesting and all the concepts can be distinguished by different sets of properties.

For example, these properties can be used to guide the search for new product measures as shown in [BMB94(b)]. Moreover, we hope this framework will help avoid future confusion, often encountered in the literature, about what properties product measures should or should not have. Studying measure properties is important in order to provide discipline and rigor to the search for new product measures. However, the relevancy of a property to a given measure must be assessed in the context of a well defined measurement concept, e.g., one should not attempt to verify if a length measure is additive.

This framework does not prevent useless measures from being defined. The usefulness of a measure can only be assessed in a given context (i.e., with respect to a given experimental goal and environment) and after a thorough experimental validation [BMB94(b)]. This framework is not a global answer to the problems of software engineering measurement; it is just of the necessary components of a measure validation process as presented in [BMB94(b)].

Future research will include the definition of more specific measurement frameworks for particular product abstractions, e.g., control flow graphs, data dependency graphs. Also, new concepts could be defined, such as information content (in the information theory sense).

Acknowledgments

We would like to thank Wolfgang Heuser, Yong-Mi Kim, Bryan Hsueh, Oliver Laitenberger, Carolyn Seaman, and Marvin Zelkowitz for reviewing the drafts of this paper.

References

[BMB94(a)] L. Briand, S. Morasca, V. Basili, "Defining and Validating High-Level Design Metrics," CS-TR 3301, University of Maryland, College Park

[BMB94(b)] L. Briand, S. Morasca, and V. R. Basili, "A Goal-Driven Definition Process for Product Metrics Based on Properties," University of Maryland, Department of Computer Science, Tech. Rep. CS-TR-3346, UMIACS-TR-94-106, 1994. Submitted for publication.

[BO94] J. Bieman and L. M. Ott, "Measuring Functional Cohesion," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 644-657, August 1994.

[C90] D. Card, "Measuring Software Design Quality," Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.

[CK94] S. R. Chidamber and C. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.

[CS91] J. C. Cherniavsky and C. H. Smith, "On Weyuker's Axioms for Software Complexity Measures," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 636-638, June 1991.

[ESE94] Encyclopaedia of Software Engineering, Wiley&Sons Inc., 1994

[F91] N. Fenton, "Software Metrics, A Rigorous Approach," Chapman&Hall, 1991.

[F94] N. Fenton, "Software Measurement: A Necessary Scientific Basis," *IEEE Trans. Software Eng.*, vol. 20, no. 3, pp. 199-206, March 1994.

[FM90] N. Fenton and A. Melton, "Deriving Structurally Based Software Measures," *J. Syst. Software*, vol. 12, pp. 177-187, 1990.

[H77] M. H. Halstead, "Elements of Software Science," Elsevier North-Holland, 1977.

[H92] W. Harrison, "An Entropy-Based Measure of Software Complexity," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 1025-1029, November 1992.

[HK81] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.*, vol. 7, no. 5, pp. 510-518, September 1981.

[LJS91] K. B. Lakshmanan, S. Jayaprakash, and P. K. Sinha, "Properties of Control-Flow Complexity Measures," *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1289-1295, Dec. 1991.

[McC76] T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 5, pp. 308-320, Apr. 1976.

[MGB90] A. C. Melton, D.A. Gustafson, J. M. Bieman, and A. A. Baker, "Mathematical Perspective of Software Measures Research," *IEE Software Eng. J.*, vol. 5, no. 5, pp. 246-254, 1990.

[O80] E. I. Oviedo, "Control Flow, Data Flow and Program Complexity," *Proc. IEEE COMPSAC*, Nov. 1980, pp. 146-152.

[P72] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, May 1972.

[P84] R. E. Prather, "An Axiomatic Theory of Software Complexity Measure," *The Computer Journal*, vol 27, n. 4, pp. 340-346, 1984.

[S92] M. Shepperd, "Algebraic Models and Metric Validation," in Formal Aspects of Measurement (T. Denvir, R. Herman, and R. W. Whitty eds.), pp. 157-173, Lecture Notes in Computer Science, Springer Verlag, 1992.

[TZ92] J. Tian and M. V. Zelkowitz, "A Formal Program Complexity Model and Its Application," *J. Syst. Software*, vol. 17, pp. 253-266, 1992.

[W88] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1357-1365, Sept. 1988.

[Z91] H. Zuse, *Software Complexity: Measures and Methods*. Amsterdam: de Gruyter, 1991.