Goal-Driven Definition of Product Metrics Based on Properties

Lionel Briand*, Sandro Morasca**, Victor R. Basili*

* Computer Science Department
University of Maryland, College Park, MD, 20742
{lionel, basili}@cs.umd.edu

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

Abstract

Defining product metrics requires a rigorous and disciplined approach, because useful metrics depend, to a very large extent, on one's goals and assumptions about the studied software process. Unlike in more mature scientific fields, it appears difficult to devise a "universal" set of metrics in software engineering, that can be used across application environments.

We propose an approach for the definition of product metrics which is driven by the experimental goals of measurement, expressed via the GQM paradigm, and is based on the mathematical properties of the metrics. This approach integrates several research contributions from the literature into a consistent , practical and rigorous approach.

The approach we outline should not be considered as a complete and definitive solution, but as a starting point for discussion about a product metric definition approach widely accepted in the software engineering community. At this point, we intend to provide an intellectual process that we think is necessary to define sound software product metrics. A precise and complete documentation of such an approach will provide the information needed to make the assessment and reuse of a new metric possible. Thus, product metrics are supported by a solid theory which facilitates their review and refinement. Moreover, their definition is made less exploratory and, as a consequence, one is less likely to identify spurious correlations between process and product metrics.

1. Introduction

Metrics can help address some of the most critical issues in software development and provide support for planning, monitoring, controlling and evaluating the software process. However, past approaches for designing new software metrics very seldom addressed a specific objective explicitly,

---

This work was supported in part by NASA grant NSG–5123, UMIACS, and NSF grant 01-5-24845. Sandro Morasca was also supported by grants from MURST and CNR.

and were usually not based upon assumptions/information about the characteristics of the development environment under study. These include descriptions of organizational structure and work procedures, guidelines, standards, etc. This frequently led to some degree of fuzziness in the metric definitions, properties, and underlying concepts, making the use of the metrics difficult, their interpretation hazardous, and the results of the various validation studies somewhat contradictory [IS88, K88].

As a consequence, the number of available metrics in the literature is quite large, but the number of used and useful metrics in industry is small. It is our position that, in order to make software measurement a viable part of the solutions to software engineering issues, metrics must be defined according to clear assumptions about the process under study and an explicit definition of the specific goal(s) to be addressed. Based on these goals and assumptions, desirable metric properties may be identified and used to direct and constrain the search for metrics. *Such an approach appears particularly necessary for product metrics since these metrics are often more complex than process metrics and address phenomena that are poorly understood.*

The goal of this paper is to specify (based on our experience [BMB93, BBH93, BMB94(a)]) a practical metric definition approach, specifically aimed at product metrics, and usable as a practical *guideline* to design technically sound and useful metrics. The focus will be the construction of prediction systems, which is a crucial application of measurement. Not all activities in this approach can, at this point, be fully formalized, nor do we believe that they will be completely formalized in the future. We think that formal techniques can be very effective in providing support for better understanding and analyzing software processes and products—indeed, we advocate the need for a formal definition of metrics' mathematical properties. However, the definition of a metric is a very human-intensive activity, which cannot be described and analyzed in a fully formal way. We believe that our metric definition approach may be better detailed, refined, and tailored to fit the needs of different application contexts. This will be made possible through the experience gained by using this metric definition approach across several environments. Thus, this work should be considered as a contribution towards a satisfactory solution. We point out what information ought to be provided when one proposes a new metric in order to make its review and refinement possible. Furthermore, we determine what intellectual process one should go through to ensure the technical soundness and practical usefulness of the defined metrics. A purely exploratory approach to metric definition would have for a consequence the experimental evaluation of a large number of relationships between product metrics (possibly not supported by any theory) and development process characteristics (e.g., effort). A simple probability calculation [F91] shows that this kind of approach is likely to lead to the identification of spurious statistical relationships, e.g., correlations uniquely due to coincidence.

Several important research issues involved in the definition of such an approach have already been investigated. Basili et al. [B92] [BR88] have provided templates to define operational experimental goals for software measurement. Melton et al. have studied product abstraction properties [MGB90]. Weyuker [W88] and Tian and Zelkowitz [TZ92] have studied desirable properties for complexity metrics. In addition, the latter authors provided a property-based classification scheme for such metrics. Fenton and Melton [FM90], and Zuse [Z90] have investigated the use of measurement theory to determine measurement scales. Finally, Schneidewind has proposed a validation framework for metrics [S92]. All this research needs to be integrated into a consistent and practical metric definition approach.

The paper is organized as follows. In the next section, we provide an overview of a practical metric design approach in part inspired by the work referenced above and augmented with some new ideas. Then, in the subsequent sections, we separately show each step of our metric design approach in detail (Sections 3-8). Section 9 outlines the directions for future work.

2. Overview of Our Metric Definition Approach

We provide here an overview of the steps composing this approach, as illustrated in Figure 1 by a Data Flow Diagram. The remaining sections will go in detail through all the issues involved in each of the steps and will provide examples.

Step 1: Define Experimental Goal(s)

Define the experimental goal(s) of the data collection, based on the general corporate objectives (e.g. reduce cycle time) and the available information about the studied development environment (e.g., weaknesses, problems). This step requires goal definition techniques. The Goal/Question/Metric paradigm (GQM) [B92] [BR88] is one of the approaches that can be used to this end. It provides a set of templates to define experimental goals and refines them into concrete and realistic questions, which subsequently lead to the definition of metrics. For instance, a GQM goal is:

Analyze *software components* for the purpose of *prediction* with respect to *the number of faults* from the viewpoint of *the project manager*.

(We will use this very simple example to illustrate the steps of our approach during this concise overview.) A GQM goal specifies the object(s) of study (*software components*), the purpose of measurement (*prediction*), the quality focus of interest *(the number of faults)*, and viewpoint (*project manager)* from which measurement is performed. The goal strongly impacts all other steps of the metric definition approach and the information they need. For instance, the object of study and the viewpoint are used to determine the product artifacts and information to be taken into account. The GQM paradigm uses descriptive models (e.g., definition of complexity metrics) and predictive models (e.g., cost models) in order to achieve the experimental goals it specifies. However, the GQM paradigm does not specify how to generate these models. In this paper, we expand the GQM paradigm to address this issue with respect to product descriptive models. As we will see in Section 6, questions about product characteristics are no longer necessary in our approach. However, GQM questions on the confidence with which assumptions are stated and on the quality (e.g., accuracy of collection procedures, granularity) of data to be collected [B92, BR88] still need to be asked. We will not address this issue, which is beyond the scope of this paper.

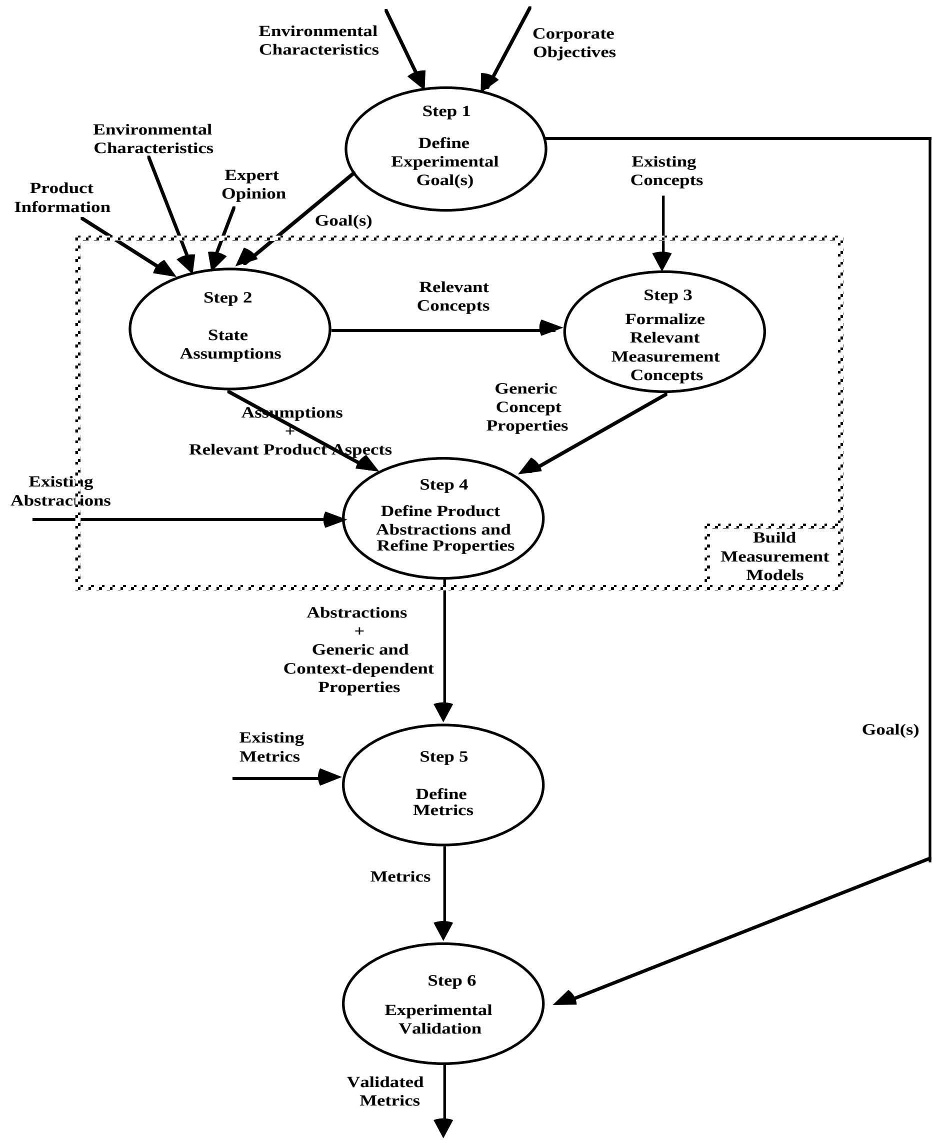
Environmental Characteristics
Corporate Objectives
Step 1
Define Experimental Goal(s)
Environmental Characteristics
Product Information
Expert Opinion
Goal(s)
Existing Concepts
Step 2
State Assumptions
Relevant Concepts
Step 3
Formalize Relevant Measurement Concepts
Assumptions + Relevant Product Aspects
Generic Concept Properties
Existing Abstractions
Step 4
Define Product Abstractions and Refine Properties
Build Measurement Models
Abstractions + Generic and Context-dependent Properties
Existing Metrics
Step 5
Define Metrics
Goal(s)
Metrics
Step 6
Experimental Validation
Validated Metrics

Figure 1. Goal-Driven and Property-Based Definition Approach for Product Metrics

Step 2: State Assumptions

Based on the object of study and the quality focus (as defined by the experimental goals, Step 1), a set of relevant assumptions must be stated to embody our intuitive knowledge about the development environment and object(s) of study. Assumptions implicitely define an order on the set of objects of study with respect to the quality focus [MGB90]. For instance, components are ordered with respect to their error-proneness. Furthermore, while stating these assumptions, relevant measurement concepts are identified, e.g., size. For instance, based on developers' interviews and a careful study of the development environment, we might assume that the larger the number of sequential blocks of statements and conditional statements in a program, the higher the number of faults. From this assumption, size appears to be a possibly relevant *measurement concept*. As an input for this step, we need information on the development environment (e.g., descriptive process model), product information and expert opinion as an intuitive basis for the assumptions. Besides assumptions, the outputs of this step also include

- a set of relevant measurement concepts (e.g., size)
- a better definition of the relevant aspects of the object of study (e.g., statement blocks and control flow)

Step 3: Formalize Relevant Measurement Concepts

Relevant *measurement concepts* of interest are formally defined (e.g., size, complexity, coupling, cohesion) through their mathematical properties. Thus, they are clearly characterized and the search for metrics is guided and constrained by these generic properties. This makes the search for metrics less exploratory and provides precise mathematical criteria for assessing the soundness of the metrics to be defined. The mathematical properties characterizing the concepts are identified independently from the concept instantiation into a metric [TZ92] [Z90] [W88] and are therefore referred to as *generic concept properties*. With reference to our simple example, we can say that a property of size is that it is non-negative. As opposed to other papers on the subject, we believe that these properties are subjective even though some of them might be widely accepted. However, it appears that, for a matter of convenience, a universal set of properties should be defined for the most important concepts used by the software engineering community, as is the case for more mature engineering disciplines. It is important, when defining metrics, that one precisely determines the meaning of concepts like size or complexity. Existing definitions may, however, be reused when available and, conversely, the newly created concepts may be stored so that they may be eventually reused.

Step 4: Define Product Abstractions and Refine Properties

One needs to define abstractions of the object of study that capture all the information (i.e., objects, attributes, relationships) needed to express the assumptions and the relevant product aspects they refer to. Some examples of product abstractions are data flow graphs, data dependency graphs, and control flow graphs. These abstractions will be representations of the object of study that will help us express useful properties and define metrics. For our example, we may assume that control flow graphs are suitable abstractions with respect to the set of assumptions and the concepts defined.

Once useful abstractions are defined, a set of new properties is added to the generic concept properties. The objective is to formalize the assumptions stated in Step 3: The intuitive ordering of the objects of study (e.g.,components) with respect to the quality focus (e.g., components' error-proneness) must be preserved by the ordering of abstractions (e.g.,components' control flow graphs) with respect to each measurement concept (e.g., components' size) [MGB90]. For instance, under the assumption stated in Step 3, and given two control flow graphs G1 and G2, we can preserve the intuitive ordering captured by the assumption if we define the following size property: the size of G1 is greater than the size of G2 if G1 has more nodes than G2. These additional properties allow us to tailor the generic concepts to any particular quality focus and set of assumptions. It should be noted that the added properties must be consistent with the generic properties defined in Step 2. These added properties are specific to a given context of measurement (i.e., goal, concept, assumptions, abstractions) and are referred to as *context-dependent properties*. At this point, if the defined abstractions are not fully adequate to define the context-dependent properties, this step can be reiterated.

Steps 2, 3, and 4, taken as a whole, can be seen as a macro-step in which *measurement models* [F91] (i.e., abstractions and generic/context-dependent properties, main outputs of Step 4) are defined based on the experimental goals, environmental characteristics, and product information (inputs of Step 2).

Step 5: Define Metrics

Metrics are defined based upon the defined product abstraction(s), concepts and their associated properties. Existing metrics can also be reused if they satisfy the defined properties. With respect to our example, size can be simply measured as the number of nodes in a control flow graph. We are not able, at this point, to select optimal metrics from those metrics satisfying the generic and context-dependent properties. Experimental validation (Step 6) will help us do so. Metrics are also generated to answer the GQM questions produced in Step 4.

Step 6: Experimental Validation

After defining metrics in Step 5, the data collected on the actual products must be used to validate the assumptions upon which the metrics are built. The procedure to follow for experimental validation varies significantly depending on the purpose of measurement. With respect to prediction, which is our main focus here, one needs to validate the product metrics with respect to their statistical relationship to the quality focus of interest. For example, we might find a very strong correlation between the defined size metric and a simple descriptive model of error-proneness, e.g., the number of faults. If the assumptions are not supported by the experimental results, we need to repeat from Step 2, re-consider the assumptions and properties, then re-define new metrics. The definition and validation of metrics are performed iteratively until the metric validation yields satisfactory results [S92].

It is important to mention that most of the outputs (e.g., product abstractions, assumptions) of the steps defined above are reusable. They should be packaged and stored so that they can be efficiently and effectively reused [BR88]. In a mature development environment, inputs for most of those steps should come from reused knowledge.

Moreover, many refinement loops are not represented in Figure 1. For example, as we said in the description of Step 6, poor experimental results may trigger the need for refining assumptions. This is an important issue that needs further investigation.

In the remainder of this paper, we will use this definition approach to define data flow size and complexity metrics as simple examples. Each step will be discussed in detail in a different section. Each section contains three subsections:

- Definition of the step
- Examples
- Discussion of related issues.

3. Define Experimental Goal(s) (Step 1)

Definition

In this section, we apply the first step of the Goal/Question/Metric paradigm [B92, BR88] to set the measurement goals. Here is a summary of templates that can be used to define goals:

Object of study: products, processes, resources
Purpose: characterization, evaluation, prediction, improvement, ...
Quality focus: cost, correctness, defect removal, changes, reliability, ...
Viewpoint: user, customer, manager, developer, corporation, ...

A detailed description of the GQM paradigm is beyond the scope of the paper. A comprehensive description of the GQM paradigm can be found in [B92, BR88].

It is important to note that the four goal dimensions mentioned above have a direct impact on the remaining steps of the metric definition approach and, from a more general perspective, the whole data collection program. This can be summarized as follows:

The *object of study* helps determine the

- software artifacts that are to be modeled by mathematical abstractions in order to be analyzable (Step 4).

- assumptions (Step 2) that may be relevant because related to the object of study.

The *purpose* points out what is the intended use of the metrics to be defined and therefore the

- type of data to be collected, e.g., process improvement requires additional data over process prediction (e.g., with respect to development effort), in order to allow for the determination of optimal techniques and methods. For example, performance data are needed in sufficient amount to ensure a minimal level of confidence in the improvement decisions.
- amount of data to be collected, e.g., if prediction usually requires more data than characterization so that the identified relationships are statistically significant. Characterization only requires the data to be representative of what is to be characterized.

The *quality focus* helps determine the

- dependent variable against which the defined product metrics are going to be experimentally validated (Step 6) [S92]. This dependent variable will in fact be a descriptive model of the quality focus. For instance, number of requirement changes per month per thousand of lines of code is a descriptive model of requirement instability. Since there may be alternative models, validation may require the use of several dependent variables. In this case, if inconsistent experimental results are obtained, the dependent variables are very likely to actually capture different quality focuses.
- assumptions (Step 2) linking the object of study characteristics to the quality focus of interest.

The *viewpoint* helps determine

- the point in time at which characterizations, predictions, or evaluations should be carried out and therefore what product information will be available to define product abstractions and metrics (Steps 4, 5).
- what information is costly or difficult to acquire and consequently, what information should be left out of the model if it does not show a sufficiently strong impact on the quality focus (Steps 5, 6).
- the definition of descriptive models of the quality focus. For example, from the user's point of view, error-proneness may be defined as the mean time to failure, whereas, from the tester point of view, it may be defined as the number of errors occuring during the test phase.

In this framework, we will not derive questions from goals as suggested by the GQM paradigm. A justification will be provided in Section 6.

Example of a goal

Let us assume that one of the corporate objectives is to reduce development time, and more particularly the time spent on testing activities. Assuming that previous studies have shown that errors are usually concentrated in a small number of "difficult" components (example of information about the development environment), the following experimental goal seems pertinent. By identifying error-prone components, we may concentrate verification activities where needed and, thereby, reduce effort.

Goal G

Object of study: component
Purpose: prediction
Quality focus: error-proneness
Viewpoint: tester

Let us take an example to illustrate the impact of the defined experimental goal on our metric definition approach. We know from the *object of study* that we have to define relevant component mathematical abstractions so we can derive component metrics. We know from the *purpose* of measurement that we need to collect enough data about the quality focus to allow a statistically significant validation of the relationships between the component metrics to be defined and the quality focus. This requires that we better define our quality focus: error-proneness. Very likely, we need to determine precisely how to count defects, e.g., what testing and inspection phases should be taken into account?, are all errors equal or should they be weighted according to a predefined error taxonomy? Such questions are also dependent on the particular *viewpoint*. In our example, testers want to find out where errors are and more particularly critical errors (according to their own definition of criticality). Therefore, errors will be weighted according to the level of criticality of their consequences. Similarly, errors could be weighted according to the correction effort they require. The determination of suitable error counting procedures will depend on the particular application of the predictive model to be built and therefore on the viewpoint of our experimental goal.

In the next sections, we will discuss more precisely about the impact of experimental goals on the definition of software product metrics.

Discussion

The definition of the goals is a fundamental phase, since all other steps in our approach are affected by the experimental goals. Therefore, extra care must be used when setting the goals. Specific descriptive process models and knowledge acquisition techniques can be used to better understand the issues that are most relevant to software development in a software organization. Careful application of the GQM paradigm provides two important results:

- Data collection is ensured to respond to the specific needs of the software organization;
- The derivation of metrics from explicit goals and the definition of explicit measurement models (output of Step 4 of our approach) allow the analyst to specify *a priori* the interpretation mechanisms associated with the collected data. This prevents *a posteriori* search for patterns which are not based on precise assumptions.

4. State Assumptions (Step 2)

Definition

We have to state assumptions (see examples below) about some aspects of the software process under study that are relevant to the experimental goals. These assumptions capture our intuitive understanding of the studied phenomena and need to be explicit so they can be discussed, questioned and refined. Various sources of information can be used to devise pertinent assumptions. A thorough understanding of the working procedures, methodologies and techniques used in the studied development environment, combined with the interview of domain experts, is usually very helpful [BBK94]. The set of assumptions defines an ordering on the set of products [MGB90] with respect to the quality focus. This ordering will be used to evaluate the adequacy of the metrics defined in the remainder of this approach.

An *assumption* is a statement believed to be true about the relationship between the *quality focus* and the characteristics of the *object of study*.

Stating assumptions helps identify the measurement concepts (e.g., size, complexity) that are characteristics of the object of study relevant to the goal. In addition, assumptions allow us to identify artifacts, or parts of artifacts (e.g., definitions, condition expressions), that must be taken into account for the definition of suitable product abstractions.

Examples of assumptions

In order to capture our intuitive understanding about data flow size and complexity, we define the following assumptions.

Assumption 1:
The larger the number of definitions and condition expressions, the larger the likelihood of error.

Assumption 2:
The larger the number of definitions and condition expressions "depending" on a definition D, the larger the probability of ripple effects if D is to be created or modified.

Assumption 3:
The larger the number of definitions on which a definition or a condition expression D "depends", the more difficult it is to create and understand D.

Assumption 4:
The larger the "distance" between two definitions or condition expression D1 and D2, where D2 depends on D1, the more difficult the control of ripple effects on D2 if D1 is to be created or modified.

The concepts between quotes are not defined: they make sense on an intuitive level. They will be formally defined later, either via the definition of product abstractions (as is the case of "dependency"), or additional concept properties in Step 4 (as is the case of "distance").

Discussion

At this point, several sets of consistent assumptions could be defined. This would lead to multiple categories of metrics, reflecting the inherent uncertainty associated with the assumptions. In Step 6, experimental results will eventually help us select the best category of metrics for each concept. For example, we could assume that when a condition expression CE (as opposed to a definition) depends on a definition D, this increases the probability of misunderstanding and ripple effect between D and CE. This stems from the fact that condition expressions also have an implicit effect on the definitions in the block they control. This additional assumption (referred to as Assumption 5) affects the metric definition approach, as we show in the following steps.

5. Formalize Relevant Measurement Concepts (Step 3)

Definition

The relevant measurement concepts are defined by specifying the mathematical properties that are believed to characterize them. In our framework, these properties should be used to constrain and guide the search for new metrics. In addition, as shown in [BMB94(b)], intuition may lead to properties showing awkward mathematical properties[1]. One should always make sure that a metric exhibits properties that are essential for its technical soundness. These properties are independent from both any specific product abstraction and any future instantiation of the concept into any specific metric. Therefore, they are called generic.

[1]The authors of this paper were several times misled in the definition of software metrics that were intuitively appealing, but, after a more thorough analysis, showed inconvenient and unsubstantiated properties.

A *measurement concept* is a class of metrics characterized by a set of mathematical properties (i.e., *generic concept properties*) and associated with an intuitive software product characteristic, e.g., size.

The generic properties associated with a measurement concept should not be contradictory—there must be at least one metric that satisfy them. Moreover, these properties should hold for the admissible transformations [Z90] of the scale of measurement (i.e., nominal, ordinal, interval, ratio, absolute) on which it is intended to define metrics. In other words, there should not be any contradiction between the scale of measurement which is assumed while using and interpreting a defined metric and its generic properties.

Examples of concepts and their generic properties

In this example, we provide properties that are, in our opinion, generic for metrics related to size and complexity. These concepts are believed to be relevant with respect to many experimental goals and applications, and in particular with respect to the goal defined above. As for complexity, the properties we define are related to the properties several authors have already provided in the literature (see [LJS92, TZ92, W88]). However, since we may want to use these properties on artifacts other than software code and on abstractions other than control-flow graphs, we formalized them in a more general manner. A thorough discussion of these properties—which is beyond the scope of this paper—can be found in [BMB94(b)]. These properties are provided as an example. Nevertheless, in the metric definition approach we outline in this paper, other sets of properties [TZ92] [W88] may be used, since the selection of properties is, to some extent, subjective.

Size and complexity are concepts related to systems, in general, i.e., one can speak about the size of a system and the complexity of a system. In our general framework—recall that we want these properties to be as independent as possible from any specific product abstraction—, a system is characterized by its elements and the relationships between them.

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 $R_m \subseteq R$. This will be denoted by $m \subseteq S$.

◊

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 fragment or a subprogram.

Concept: Size

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 2: 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 = <E,R> is non-negative

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

◊

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

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

◊

Property *Size.3*: Module Additivity
The size of a system S = <E,R> 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

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

◊

The last property Size.3 provides the means to compute the size of a system S = <E,R> 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[2].

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

Therefore, adding elements to a system cannot decrease its size

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

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

From the above properties Size.1 - Size.3, it also 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.,

$(m_1 \subseteq S$ **and** $m_2 \subseteq S$ **and** $E = E_{m1} \cup E_{m2}) \Rightarrow \mathrm{Size}(S) \leq \mathrm{Size}(m_1) + \mathrm{Size}(m_2)$
(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 metrics on a ratio scale.

Concept: Complexity

Intuitively, the complexity of a product is a measurement concept that is considered extremely relevant to system properties. It has been studied by several researchers [BMB94(b)]. In our framework, we expect product 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 [BMB94(b)].

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 metric 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 relationships (note $R' \cup R'' \subseteq R$ in formula Complexity.IV's premise) between the elements of each module (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).

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.

Definition 3: 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

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

◊

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

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

$(S = \langle E,R \rangle$ **and** $S^{-1} = \langle E,R^{-1} \rangle) \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1})$
(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

$$(S = \langle E,R \rangle \textbf{ and } m_1 = \langle E_{m1},R_{m1} \rangle \textbf{ and } m_2 = \langle E_{m2},R_{m2} \rangle$$
$$\textbf{and } E_{m1} \cup E_{m2} \subseteq E \textbf{ and } R_{m1} \cup R_{m2} \subseteq R \textbf{ and } R_{m1} \cap R_{m2} = \emptyset)$$
$$\Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2)$$
(Complexity.IV)

◊

Property *Complexity*.5: Disjoint Module Additivity
The complexity of a system $S = \langle E,R \rangle$ composed of two disjoint modules $m_1 = \langle E_{m1},R_{m1} \rangle$, $m_2 = \langle E_{m2},R_{m2} \rangle$ is equal to the sum of the complexities of the two modules

$$(S = \langle E_{m1} \cup E_{m2},R_{m1} \cup R_{m2} \rangle \textbf{ and } E_{m1} \cap E_{m2} = \emptyset \textbf{ and } R_{m1} \cap R_{m2} = \emptyset)$$
$$\Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2)$$
(Complexity.V)

◊

As a consequence of the above properties Complexity.1 - Complexity.5, it can be shown that the complexity of a system is no less than the complexity of any of its modules, i.e., adding relationships between elements of a system does not decrease its complexity

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

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

Discussion

The paragraphs above, stating the motivations and justifications for size and complexity concepts, illustrate the subjectivity of the metric definition approach. However, it is important that all concept properties be explicitly justified and motivated so that their limitations may be understood and the discussion on their validity may be facilitated.

6. Define Product Abstractions and Refine Concept Properties (Step 4)

Definition

We first need to define an abstraction that helps us precisely capture and define all the concepts involved in the stated assumptions. Abstractions are mathematical representations of the product(s) (usually graphs). Products have to be mapped into abstractions so they become analyzable and some of their attributes become quantifiable [MGBB90]. The choice should be entirely guided by the experimental goals (i.e., the object of study and the quality focus) and the set of assumptions, that is, the abstractions must capture all the concepts involved in the set of assumptions related to the object of study. The mapping from the product to the abstraction needs to be checked for completeness, i.e., Does the abstraction contain all the relationships between nodes that one wants to capture? Is the level of granularity of the abstraction nodes sufficient to represent accurately the product? One way of assessing the suitability of an abstraction is to study the effect of relevant modifications in the product and assess its impact on the abstraction, e.g., number of nodes and edges added or removed, change of topology in a graph. Several abstractions capturing control flow, data flow and data dependency information are available in the literature [M90, BBC88, O80]. However, an even larger variety of abstractions can be derived from software products.

The set of properties associated with each concept is expanded so as to formalize the order existing on the set of abstractions with respect to each concept as defined by the assumptions. Therefore, the order formalized by the newly introduced properties is intended to preserve the order defined by the assumptions so that concepts have a monotonic relationship with the quality focus of interest. For example, given that the quality focus is error-proneness and that a Definition-Use (D-U) graph DUG1 is defined as more complex than another graph DUG2 and assuming that there is a monotonic relationship between error-proneness and complexity, we expect the assumptions to state that the product corresponding to DUG1 is more error-prone than that of DUG2.

These properties are specific to a given context of measurement (i.e., goal, concept, assumptions, abstractions) and are referred to as *context-dependent properties*. They will, most of the time, capture effects on the ordering of abstractions when modifications are performed on these abstraction. These modifications will often be what is referenced as atomic modifications in [Z90], adding / removing / moving / substituting an edge/node. They will be useful in order to constrain and guide the search for metrics (Step 5).

Examples

In our example, D-U graphs are a suitable abstraction since they capture concepts such as definitions, condition expressions, uses. D-U graphs are directed graphs where nodes are statements or conditions and arcs are definition-use clear paths [RW82]. Moreover, concepts such as "dependencies" or "distance" can be derived from such graphs. A definition or a condition expression "depends" on a definition when the variable/constant defined in the latter is used in the former. A suitable definition of "distance" between two definitions will be provided in the next section.

Concept: Size

Property *CD1*: Count of definitions
If a graph DUG_1 has at least as many definitions and condition expressions as another graph DUG_2, then $Size(DUG_1) \geq Size(DUG_2)$.

◊

The above property CD1 is not implied by the generic properties Size.1-Size.3, since DUG_1 and DUG_2 have nothing to do with each other, i.e., they are not related by any inclusion relationship (DUG_2 is not necessarily included in DUG_1).

Concept: Complexity

Property *CD2*: Sum of distances
Let DUG_1 and DUG_2 be two Definition-Use graphs. If the sum of the distances between all pairs of nodes in DUG_1 is greater than the sum of distances between all pairs of nodes in DUG_2, then $Complexity(DUG_1) > Complexity(DUG_2)$.

◊

The distance between two nodes is the number of arcs in the longest path between the two nodes that contains no repetitions of elementary cycles (cycles that do not traverse the same arc twice). As an example, the distance between nodes b and c in the D-U graph of Figure 2 is 4, i.e., the number of arcs of the path {<b,c>,<c,e>,<e,b>,<b,c>}. In this path, the arc <b,c> is traversed twice, but it is only traversed once in the cycles {<b,c>,<c,e>,<e,b>} and {<c,e>,<e,b>,<b,c>} contained in the path. When several paths exist between two nodes, we select the longest one because the shortest or average path distance would not satisfy the monotonicity property (Complexity.4). For instance, adding an arc in a graph may decrease the length of the shortest path between two nodes. The distance between two unrelated nodes is zero because the absence of relation does not add any complexity, consistent with the generic property Complexity.2. This shows how generic properties constrain the definition of metrics and help make the right decisions. As an example of distance calculations, consider the D-U graph in Figure 2.

If Assumption 5 is considered, a different abstraction is necessary: Data-Dependency (D-D) graphs [BBC88]. This abstraction captures the links between condition expressions and the definitions they can affect. In this case, the following property holds:

Property *CD3*: Definitions versus condition expressions
Let DDG_1 and DDG_2 be two Data Dependency graphs. If DDG_2 is identical to DDG_1 except for the fact that one of the condition expressions of DDG_1 has been substituted with a definition to form DDG_2, then $Complexity(DDG_1) > Complexity(DDG_2)$. In other words, a condition expression is the source of more complexity than a definition.

◊

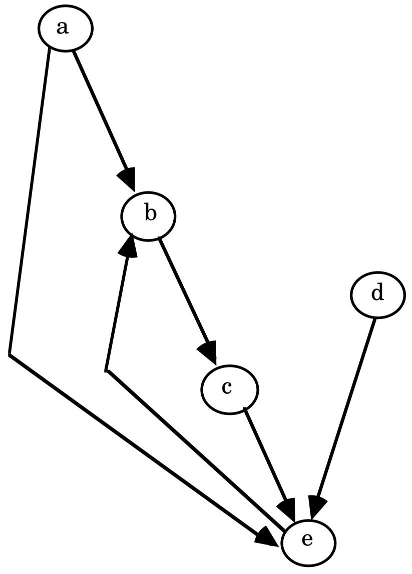

Figure 2. Example of D-U graph

The distances between the nodes in Figure 2 are computed in Table 1.

	a	b	c	d	e
a	0	5	6	0	6
b	0	3	4	0	5
c	0	5	3	0	4
d	0	5	6	0	4
e	0	4	5	0	3

Table 1. Distances between the nodes of the D-U graph in Figure 2

Discussion

According to the GQM paradigm, questions must be derived from goals. In our particular framework, questions about product characteristics (e.g., what is the complexity of a component?) are not necessary and the outputs of Steps 2, 3, and 4 may be seen as a more rigorous substitute to questions. Thus, metrics are not intended to answer questions but to validate assumptions. However, as we have shown, there may be aspects of the relevant environmental characteristics that cannot be explicitly modeled, e.g., the quality of the data and the validity of the assumptions, so questions may still be necessary to support the full interpretation of the metrics.

As pointed out in [FM90, F94], not all abstractions may be comparable with respect to a particular measurement concept. In such cases, it appears difficult to define a total order on the set of abstractions and only a partial order can be obtained [MGB90]. Ultimately, statistical analysis can only be conducted independently on comparable subsets of abstractions.

One of the main difficulties of this step is to ensure that the set of context-dependent properties is complete. Completeness is reached when the properties can fully describe the ordering of abstractions, i.e., when any pair of comparable abstractions can be ordered by using the stated properties or their combination.

It is also necessary to verify that the newly introduced context-dependent properties define metrics whose scales are consistent with those defined by the generic properties, i.e., ratio, interval, ordinal, nominal.

7. Define Metrics (Step 5)

Definition

For each concept, metrics are defined by using the abstractions' elements and relationships and are checked against the concepts' generic and context-dependent properties. Management and resource constraints are taken into account at this point for defining convenient metrics. This step may require approximations which must be performed explicitly, based on a solid theory, and in a controlled manner. At this stage, we are not able to select the best among alternative metrics satisfying generic and context-dependent properties. Experimental validation (Step 6) will help us perform such a selection. As a necessary precondition to carrying out a meaningful experimental validation, the measurement scale (i.e., nominal, ordinal, interval, ratio, absolute [FM90], [Z90]) of the metrics must be clearly identified. This prevents metrics from being misused (e.g., taking the average value of an ordinal metric, which is meaningless).

Examples

Concept: Size

A simple size metric is given by the number of definitions and condition expressions, i.e., the number of nodes in the Definition-Use graph. Other size metrics can be devised, by associating a weight with each node. However, this would require that additional assumptions be made.

Concept: Complexity

The most straightforward metric that comes to mind is the number of arcs in the graph. However, this does not take into account Assumption 4 since distances between pairs of nodes may not have an impact on the metric. In this context, a complexity metric that seems relevant and that satisfies the generic and context-dependent properties is the sum of distances between every pair of nodes in the DUG graph.

$$\mathrm{Cplx(DUG)}=\sum_{i=1}^{|E|}\sum_{j=1}^{|E|}\mathrm{Distance}(\mathrm{Node}_i,\mathrm{Node}_j) \qquad (1)$$

where $\mathrm{Node}_i, \mathrm{Node}_j \in E$.

If Assumption 5 and Property CD3 are taken into account, then another complexity metric can be defined as follows

$$\mathrm{Cplx(DDG)}=\sum_{i=1}^{|E|}\sum_{j=1}^{|E|}\mathrm{Distance}(\mathrm{Node}_i,\mathrm{Node}_j) \qquad (2)$$

Note that the formula is identical but the abstraction used is different, i.e., Data-Dependency Graphs (DDG). This metric is therefore different from the one in (1). The weight of condition expressions in formula (2) has increased since path distances are made longer by the link between condition expressions and the definitions that belong to the block they control.

Discussion

Once metrics have been defined, it must be proven that they are consistent with the generic and context-dependent properties. With reference to our examples, it can be easily shown that the metrics we define for size and complexity satisfy their respective sets of generic and context-dependent properties. Thus, they can be shown to preserve the intuitive order defined on the abstractions with respect to the quality focus.

8. Experimental Validation of the Metrics (Step 6)

After defining metrics in Step 5, the data collected on actual software products and processes must be used to validate the metrics experimentally. This is done differently according to the purpose of measurement. With respect to prediction, it is required to validate the assumptions on which the product metrics are based. In other words, significant statistical relationships must be identified between the product metrics and the quality focus (or rather a particular descriptive model of the quality focus) and, furthermore, these relationships must be consistent with what is specified by the assumptions. Validation procedures for other measurement purposes (e.g., characterization) will not be discussed here.

With respect to prediction, experimental validation may be seen as a search for statistical relationships between metrics of the *object of study* and a descriptive models of the *quality focus* (e.g., # error for Error-proneness).

Numerous analysis techniques, both univariate and multivariate [S92, BBH93, DG84], exist in the statistical and machine learning literature. If such assumptions and properties are not validated, we need to repeat from Step 2, re-consider the assumptions and properties, then re-define new metrics. This metric definition/validation cycle is iterated until the metric validation yields satisfactory results. Since extensive material is available on the subject, we will not describe this step any further.

9. Conclusions and Future Work

Product metrics need to be defined in a rigourous and disciplined manner based on a precisely stated experimental goal, assumptions, properties, and a thorough experimental validation. In order to do so, we propose a definition approach that is intended to help analysts develop product metrics. This approach integrates many contributions from the literature and is intended to be the starting point for a practical product metric definition approach to be discussed by the software engineering community, on both the academic and industrial sides. This approach is the result of our past experience [BMB93, BBH93, BMB94(a)] and is validated through realistic examples.

Our future work encompasses a more detailed study and validation of each of the steps involved in the metric definition approach. In this framework, we proposed definitions for the measurement concepts usually encountered in software engineering, such as complexity, size, coupling, cohesion, etc [BMB94(b)]. Such a work aims at building a formal, unambiguous, and comprehensive theory. Also, we need to better understand how experimental results can be used to guide the refinement of metric. The refinement process of metrics needs to be better understood and defined so that metrics can evolve with the increase in understanding and refinement of the studied development processes. Last, we need to better identify what can be reused across environments and projects, e.g., metrics, assumptions, measurement concepts, product abstractions.

Acknowledgments

We would like to thank William Agresti, Dieter Rombach, Yong-Mi Kim, Bryan Hsueh, Wolfgang Heuser, Oliver Laitenberger, and Manoel Mendonça for their help in reviewing the early drafts of this paper.

References

[B92] V. Basili, "Software Modeling and Measurement: The Goal/Question/Metric Paradigm" University of Maryland, Department of Computer Science, Tech. Rep. CS-TR-2956, 1992.

[BBC88] J. Bieman et al, "A Standard Representation of Imperative Language Programs for Data Collection and Software Measures Specification", *J. Syst. Software*, vol. 8, pp. 13-37, 1988.

[BBH93] L. Briand, V. Basili and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components," *IEEE Trans. Software Eng.*, 19 (11), November, 1993.

[BBK94] L. Briand, V. Basili, Y. M. Kim and D. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," IEEE Conference on Software Maintenance, September 1994, Victoria, British Columbia, Canada.

[BMB93] L. Briand, S. Morasca, V. Basili, "Assessing Software Maintainability at the End of High-Level Design", IEEE Conference on Software Maintenance, September 1993, Montreal, Quebec, Canada.

[BMB94(a)] L. Briand, S. Morasca, V. Basili, "Defining and Validating High-Level Design Metrics", CS-TR 3301, UMIACS-TR 94-75, University of Maryland, College Park

[BMB94(b)] L. Briand, S. Morasca, V. Basili, "Property-based Software Engineering Measurement," CS-TR 3368, UMIACS-TR 94-119, University of Maryland, College Park

[BR88] V. Basili and D. Rombach, "The Tame Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, pp. 758-773, June 1988.

[DG84] W. Dillon and M. Goldstein, *Multivariate Analysis: Methods and Applications*, Wiley and Sons, 1984.

[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.

[IS88] D. Ince, M. Shepperd, "System Design Metrics: a Review and Perspective," Proc. Software Engineering 88, pages 23-27, 1988

[K88] B. Kitchenham, "An Evaluation of Software Structure Metrics," Proc. COMPSAC 88, 1988

[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.

[M90] L. Moser, "Data Dependency Graphs for Ada Programs", *IEEE Trans. Software Eng.*, vol. 16, no. 5, pp. 498-509, May 1990.

[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. COMPSAC*, Nov. 1980, pp. 146-152.

[RW82] S. Rapps and E. Weyuker, "Data flow analysis test techniques for program test data selection", in *Proc. 6th Int. Conf. on Software Engineering*, Sept. 1982, pp. 272-278

[S92] N. F. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410-422, May 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.

[Z90] H. Zuse, *Software Complexity: Measures and Methods*. Amsterdam: de Gruyter, 1990.