Fundamental Aspects of Software Measurement

Sandro Morasca

Università degli Studi dell'Insubria
Dipartimento di Scienze Biomediche, Informatiche e della Comunicazione
I-22100 Como, Italy
sandro.morasca@uninsubria.it

Abstract. Empirical studies are increasingly being used in Software Engineering research and practice. These studies rely on information obtained by measuring software artifacts and processes, and provide both measures and models based on measures as results. This paper illustrates a number of fundamental aspects of Software Measurement in the definition of measures that make sense, so they can be used appropriately. Specifically, we describe the foundations of measurement established by Measurement Theory and show how they can be used in Software Measurement for both internal and external software attributes. We also describe Axiomatic Approaches that have been defined in Software Measurement to capture the properties that measures for various software attributes are required to have. Finally, we show how Measurement Theory and Axiomatic Approaches can be used in an organized process for the definition and validation of measures used for building prediction models.

Keywords: Software Measurement, Measurement Theory, internal software attributes, external software attributes, Axiomatic Approaches, GQM.

1 Introduction

Measurement is an essential part in every scientific and engineering discipline and is a basic activity in everyday life. We use measurement for a variety of goals, by acquiring information that we can use for developing theories and models, devising, assessing, and using methods and techniques, and making informed and rational practical decisions. Researchers use measurement to provide evidence for supporting newly proposed techniques or for critically assessing existing ones. Industry practitioners use measurement for production planning, monitoring, and control, decision making, carrying out cost/benefit analyses, post-mortem analysis of production projects, learning from experience, improvement, etc. Final consumers use measurement to make sensible decision.

Thus, it is not surprising that measurement is at the core of many engineering disciplines, and the interest towards measurement has been steadily growing in Software Engineering too. However, Software Engineering differs from other engineering disciplines in a number of aspects that deeply affect Software Measurement.

- Software Engineering is a relatively young discipline, so its theories, methods, models, and techniques still need to be fully developed, assessed, consolidated, and improved. The body of knowledge in Software Engineering is still limited, if compared to the majority of engineering disciplines, which have been able to take advantage of scientific models and theories that have been elaborated over the centuries. These models and theories were built through a process that required 1) the identification of a number of fundamental concepts (e.g., length, mass, time, and electrical charge in Physics), 2) the study of their characteristics, 3) the investigation of how they are related to each other by means of theories and models, and 4) how they can be measured by collecting data from the field so theories and models can be validated and used.
- Software Engineering is a very human-intensive discipline, unlike the engineering branches that are based on the so-called hard sciences (e.g., Physics, Chemistry). One of the main tenets of these sciences is the repeatability experiments and their results. This is hardly ever the case in a number of interesting Software Engineering studies. For instance, it is clearly impossible to achieve repeatability when it comes to developing a software product. So, it is virtually impossible that given the same requirements and given two teams of developers with the same characteristics working in environments with the same characteristics, we obtain two identical values for the effort needed to develop a software application. Thus, Software Measurement models, theories, methods, and techniques will be different from those of the hard sciences and will probably not have the same nature, precision, and accuracy. As a matter of fact, some aspects of Software Measurement are more similar to measurement in the social sciences than measurement in the hard sciences.
- Software Engineering deals with artifacts that are mostly immaterial, like software code, test suites, requirements, etc., for which there is no real, direct experience. For instance, we are much more acquainted with the concept of length of a material object than the concept of "complexity" of a software program.

Given these specific aspects of Software Engineering, and the practical impact that Software Measurement can have on Software Engineering practices, we here investigate two basic questions, similar to those that are frequently quoted for Software Verification and Validation.

- Are we measuring the attributes right? When using or defining a measure, it is necessary to make sure that it truly quantifies the attribute it purports to measure. As mentioned above, Software Engineering has not yet reached the same level of maturity of other engineering disciplines, nor has Software Measurement, so this theoretical validation is a required activity for using or defining measures that make sense. Theoretical validation is also

a necessary step not only for the empirical validation of software measures, but, even more importantly, for empirically validating Software Engineering techniques, methods, and theories. For instance, empirically validating the claim that the maintainability of a software system decreases if the coupling between its modules increases requires that one use sensible measures for coupling and maintainability. Given the lack of intuition about software product attributes, theoretical validation is not a trivial activity, as it involves formalizing intuitive ideas around which a widespread consensus still needs to be built.
– Are we measuring the right attributes? Measuring attributes that are irrelevant for our goals would clearly be useless. So, we need to select the right ones, given the available resources. The best way to provide evidence that an attribute is relevant for some specified goal is to use a sensible measure for that attribute and carry out an experimental or empirical study by showing that, for instance, it can be used for predicting some software product or process attribute of interest. This activity entails the empirical validation of the measure.

By not answering these two questions satisfactorily, we could end up with measures that are useless and waste the resources used for measurement. In an even worse scenario, we could actually make incorrect decisions based on inappropriate measures, which could even be detrimental to the achievement of our goals, in addition to wasting the resources used for measurement.

In the remainder of this paper, we describe the foundational aspects of Software Measurement, by using the two main approaches available in the related literature, namely Measurement Theory and Axiomatic Approaches. Beyond their mathematical appearance, both approaches are actually ways to formalize common sense, so as to reach a reasonable compromise between the principles of rigor and common sense that should rule in applied research. We describe these two approaches in the remainder of this paper to show their usefulness and potential, along with their strengths and weaknesses. In addition, we illustrate a goal-oriented process that can be followed to carry out the definition and empirical validation of sensible measures. This process also shows how the theoretical validation of measures can be used to prevent irrelevant measures or attributes from being used in prediction models.

We mostly address fundamental issues in the measurement of attributes of software artifacts in this paper. So, our primary focus here is on the theoretical validation of measures for attributes of software artifacts. The measurement of process attributes is somewhat more straightforward. Take design effort as an example of a software process attribute. The real challenge in this case is the prediction of the effort required to carry out software design, not its measurement. Software design effort prediction requires the identification of appropriate measures that can be quantified before design takes place and the availability of a quantitative model based on those measures. Software design effort measurement, instead, simply requires the identification of which activities belong to software design, the people who work on those activities, and the use of accurate collection tools for the effort used for software design. Thus, software design effort measurement is mostly concerned with a number of details which, albeit important, do not compare to the issues related to the building of a good prediction model.

The remainder of this paper is organized as follows. Section 2 introduces a few basic concepts and the terminology that is used in the paper. The use of Measurement Theory for the intrinsic attributes of software artifacts is described in Section 3, while Axiomatic Approaches are introduced in Section 4. A unified axiomatic approach for the description of a number of interesting intrinsic attributes of software artifacts is described in more detail in Section 5. Section 6 describes Probability Representations, which are a part of Measurement Theory that has received little attention in Software Measurement so far, and which can be used for providing firm foundations for all of those software artifact attributes that are of practical interest for software stakeholders. Section 7 illustrates GQM/MEasurement DEfinition Approach (GQM/MEDEA), a process for the definition of measures for prediction models that integrates the fundamental aspects of Software Measurement into a coherent, practical approach. Conclusions and an outline of future work on fundamental aspects of Software Measurement follow in Section 8.

2 A Few Basic Concepts

Here, we introduce a few basic concepts and terminology that will be used throughout the paper.

In Software Measurement, like in any kind of measurement, one measures the attributes of entities. It does not make sense to measure an "entity," without mentioning which specific attribute of that entity we would like to measure. It does not make much sense to "measure a car," for instance. It is necessary to specify whether we are measuring its length, width, height, weight, number of seats, maximum speed, etc. By the same token, it does not make much sense to measure a software program, unless we specify which particular attribute we have in mind. There is a plethora of different attributes that have been introduced in Software Measurement: size, complexity, cohesion, coupling, connectivity, usability, maintainability, readability, reliability, effort, development time, etc., to mention a few. Since there are so many of them, it is important to understand the nature of these attributes and identify their similarities and differences. In the long term, it would be useful to come to a generalized agreement about these fundamental concepts of Software Measurement so that everybody uses and understands the same concepts in the same way. For terminology consistency, we use the term attribute in this paper, instead of other terms that have been used in the past for the same concept, including quality, characteristic, subcharacteristic, factor, criterion.

Conversely, it does not make sense to measure an attribute without mentioning the entity on which it is measured. For instance, it does not make sense to measure the number of seats without referring to a specific car. In Software

Measurement, it would not make sense to measure "complexity" without specifying whose program it is. Software Measurement is used on a number of different entities that belong to the following two categories:

– software products and documents, e.g., source code, executable code, software design, test suites, requirements; we use the term "software artifact" to refer to any such entity;
– activities carried out during software production process, e.g., coding, deployment, testing, requirements analysis; we use the term "software process" to refer to any such entity.

A typical distinction in Software Measurement is made between internal and external attributes of entities.

– Internal attributes of software artifacts, such as size, structural complexity, coupling, cohesion, are usually said to be as those attributes of an entity that *can be measured* based only on the knowledge of the entity [15]. So, internal attributes are easy to measure: for instance, the size of a program is often measured by counting the number of its lines of code.
– External attributes, such as reliability, performance, usability, maintainability, portability, readability, testability, understandability, reusability, are characterized as those attributes that *cannot be measured* based only on the knowledge of the software artifact. Their measurement involves the artifact, its "environment," and the interactions between the artifact and the environment. For instance, the maintainability of a software program depends on the program itself, the team of people in charge of maintaining the program, the tools used, etc. The maintainability of a given program is likely to be higher if maintenance is carried out by the same people that developed the program than by other programmers. So, the knowledge of the program alone is not sufficient to quantify its maintainability.

From a practical point of view, external software attributes are the ones related to and of direct interest for the various categories of software "users," e.g.: the compiler/interpreter that translates a program; the computer on which it runs; the final user; the practitioners. These "users" may have different and possibly conflicting needs. The ensemble of the attributes that are relevant to the users completely describes what is known as the quality of a software product. Therefore, external attributes are the ones that have true industrial interest and relevance. However, because of their very nature, external attributes are in general more difficult to define and quantify than internal ones, as they require that a number of factors be taken into account, in addition to the software artifact [15,36]. On the other hand, internal attributes are much easier to quantify. However, they have no real interest or relevance *per se*. The measurement of an internal attribute of a software artifact (e.g., the size of a software design) is interesting only because it is believed or it is shown that the internal attribute is linked to: (1) some external attribute of the same artifact (e.g., the maintainability of the software design) or of some other artifact (e.g., the fault-proneness of the software code); (2) some attribute of the software process (e.g., the effort needed to develop the software design). So, one typically measures internal software attributes to assess or predict the value of external software attribute.

One final terminology clarification is in order before we start reviewing various fundamental aspects of Software Measurement. The term "metric" has been often used instead of "measure" in the Software Measurement and Software Engineering in the past. As it has been pointed out, "metric" has a more specialized meaning than "measure." The term "metric" is closely related to distance and it typically implies the presence of some unit of measurement. As we explain in the remainder of this paper, this is not necessarily the case, so the more general term "measure" is preferable. Therefore, we consistently use "measure" in the remainder of this paper.

3 Measurement Theory for Internal Software Attributes

The foundations of Measurement Theory were established by Stevens [38] in the 1940s, as a way to provide the mathematical underpinnings for measurement in the social and human sciences. It has been used in other disciplines and its concepts have been extended and consequences have been assessed since. Measurement Theory is now a quite well-established field. The interested reader can refer to [20,34] for more complete introductions to the subject. In Empirical Software Engineering, Measurement Theory has almost exclusively been used with reference to the measurement of the attributes of software artifacts, such as size, structural complexity, cohesion, coupling.

3.1 Basic Notions of Measurement Theory

The first and most important goal of Measurement Theory is to make sure that measures have properties that make them comply with intuition. So, Measurement Theory [20,34] makes clear that measuring is not just about numbers, i.e., assigning measurement values to entities for some attribute of interest. For instance, it would make little sense to have a software size measure that tells us that a program segment is longer than another program segment when we look at those two segments and conclude that it should actually be the other way round.

Beyond all the mathematics involved, Measurement Theory shows how to build a sensible, common sense bridge, i.e., a measure (Definition 3), between

- our "intuitive," empirical knowledge on a specified attribute of a specified set of entities, via the so-called Empirical Relational System (Definition 1), and
- the "quantitative," numerical knowledge about the attribute, via the so-called Numerical Relational System (Definition 2), so that
- the measure makes sense, i.e., it satisfies the so-called Representation Condition (Definition 4).

We now explain these concepts and we use the size of a set of program segments as an example to make these definitions more concrete.

Definition 1 (Empirical Relational System). *Given an attribute, let*

- E *denote the set of entities for which we would like to measure the attribute*
- $R_1, \ldots, R_y$ *denote y empirical relations capturing our intuitive knowledge on the attribute: each* R_i *has an arity* n_i*, so* $R_i \subseteq E^{n_i}$*; we write* $(e_1, \ldots, e_{n_i}) \in R_i$ *to denote that tuple* $(e_1, \ldots, e_{n_i})$ *is in relation* R_i*; if* R_i *is a binary relation, we use the infix notation* $e_1 R_i e_2$
- o_1*, ...,* o_z *denote z empirical binary operations on the entities that describe how the combination of two entities yields another entity, i.e.,* $o_j : E \times E \to E$*; we use an infix notation, e.g.,* $e_3 = e_1 o_j e_2$*.*

An Empirical Relational System is is an ordered tuple

$$ERS = (E, R_1, \ldots, R_y, o_1, ..., o_z)$$

For instance, suppose we want to study the size of program segments. We typically have

- the set of entities E is the set of program segments
- *longer_than* $\subseteq E \times E$, an empirical binary relation that represents our knowledge that, given any two program segments e_1 and e_2 for which e_1*longer_than*e_2, e_1 has a greater size than e_2
- a concatenation operation, i.e., $e_3 = e_1; e_2$.

Other attributes of the same set of entities, e.g., complexity, will have different kinds and sets of empirical relationships and operations than size has. This is due to the fact that we have different intuitions about different attributes of a set of entities.

No numbers or values are found in the Empirical Relational System, which only takes care of modeling our own empirical intuition. Measurement values are introduced by the Numerical Relational System, which we define next

Definition 2 (Numerical Relational System). *Given an attribute, let*

- V *be the set of values that we use to measure the attribute*
- $S_1, \ldots, S_y$ *denote y relations on the values: each* S_i *has the same arity* n_i *of* R_i
- $\bullet_1$*, ...,* $\bullet_z$ *denote z numerical binary operations on the values, so each* $\bullet_j$ *has the form* $\bullet_j : V \times V \to V$*; we use an infix notation, e.g.,* $v_3 = v_1 \bullet_j v_2$*.*

A Numerical Relational System is an ordered tuple

$$NRS = (V, S_1, \ldots, S_y, \bullet_1, ..., \bullet_z)$$

Even though it is called Numerical Relational System, We have chosen to represent V as a set of “values” and not necessarily numbers for greater generality and because in some cases numbers are not really needed and may even be misleading (e.g., for nominal or ordinal measures as described later in this section). In our segment size example, we can take

– $V = Re_{0+}$, the set of nonnegative real numbers, which means that the values of the size measures we use are nonegative real numbers
– a binary relation '$>$', which means that we want to use the natural ordering on those measurement values (so we can translate “longer_than” into '$>$' and back as the Representation Condition will mandate)
– a binary operation '$+$', which means want to be able to sum the sizes of segments (the Representation Condition will actually mandate that we sum the sizes of concatenated program segments).

The Numerical Relational System is purposefully defined to mirror the Empirical Relational System in the realm of values, even though the Numerical Relational System in itself does not predicate about the entities and the specific attribute investigated.

The connection between the Empirical Relational System and the Numerical Relational System, and thus, entities and values, is made via the concept of measure (Definition 3).

Definition 3 (Measure). *A function $m : E \rightarrow V$ is said to be a measure.*

However, there is more to the Empirical Relational System than just the set of entities on which it is based. The Empirical Relational System also gives information about what we know about an attribute of a set of entities. If that knowledge is not taken into account, any $m \in V^E$ is a measure, i.e., any assignment of values to program segments may be a measure, according to Definition 3. Given program segments e_1, e_2, e_3 such that $e_1 longer\_than e_2$ and $e_2 longer\_than e_3$, a measure m according to Definition 3 may be very well provide values of $m(e_1)$, $m(e_2)$, and $m(e_3)$ such that $m(e_1) < m(e_2)$ and $m(e_3) < m(e_2)$, though this does not make sense to us. Measurement Theory introduces the Representation Condition (Definition 4) to discard all of those measures that contradict our intuition and keep only the fully sensible ones.

Definition 4 (Representation Condition). *A measure must satisfy the two conditions*

$$\forall i \in 1 \ldots n, \forall (e_1, \ldots, e_{n_i}) \in E^{n_i} (e_1, \ldots, e_{n_i}) \in R_i \Leftrightarrow (m(e_1), \ldots, m(e_{n_i})) \in S_i$$
$$\forall j \in 1 \ldots m, \forall (e_1, e_2) \in E \times E (m(e_1 o_j e_2) = m(e_1) \bullet_j m(e_2))$$

The Representation Condition translates into the following two conditions for our segment size example

– $e_1 longer\_than e_2 \Leftrightarrow m(e_1) > m(e_2)$, i.e., our intuition on the ordering of the program segments is mirrored by the ordering of the measurement values, and *vice versa*,

- $m(e_1; e_2) = m(e_1) + m(e_2)$, i.e., the size of a program segment obtained by concatenating two program segments is the sum of the sizes of the two program segments.

A sensible measure is defined as a scale (Definition 5) in Measurement Theory.

Definition 5 (Scale). *A scale is a triple* (ERS, NRS, m)*, where* ERS *is an Empirical Relational System,* NRS *is a Numerical Relational System, and* m *is a measure that satisfies the Representation Condition.*

In what follows, we assume for simplicity that measures satisfy the Representation Condition, so we use the terms "scale" and "measure" interchangeably, unless explicitly stated.

So now, given an Empirical Relational System and a Numerical Relational System, we would need to find out if we can actually build a measure. However, the existence of a measure will depend on the specific Empirical Relational System and a Numerical Relational System, and we will not illustrate the issues related to the existence of a measure in detail. Rather, we investigate whether more than one legitimate measure may be built, given an Empirical Relational System and a Numerical Relational System. This should not come as a surprise, since it is well known from real life that we can quantify certain attributes of physical objects by using different equally legitimate measures. For instance, the length of a segment may be quantified equally well by meters, centimeters, yards, feet, inches, etc. We know that we can work equally well with one measure or another and that one measure can be translated into another by means of a multiplicative factor. For notational convenience, we denote by $M(ERS, NRS)$ the set of scales that can be defined based on an Empirical Relational System ERS and a Numerical Relational System NRS.

However, the very existence of a set of equally good measures shows something a little bit more surprising: the bare value of a measure in itself does not provide a lot of information. For instance, saying that the length of a wooden board is 23 does not mean much, unless one specifies the unit of measurement. Clearly, talking about a 23 inch wooden board is not the same as talking about a 23 meter one. Introducing the concept of unit of measurement actually means taking one object as the reference one and then assigning a measurement value to all other objects as the times the other objects possess that specified attribute with respect to the reference object. So, if we say that a wooden board is 23 inches long, all we are saying is that it is 23 times longer than some wooden board that we took as the one that measures 1 inch. What is even more important is that this 23:1 ratio between the length of these two wooden boards is the same no matter the measure used to quantify their length, be it inches, feet, meters, etc. This is true for the ratios of the lengths of any pair of wooden boards, i.e., these ratios are invariant no matter the scale used. Invariant properties of scales are called meaningful statements and provide the real information content of a scale, as they do not depend on the conventional and arbitrary choice of one specific scale.

Definition 6 (Meaningful Statement). *A statement $S(m)$ that depends on a measure m is* meaningful *if its truth value does not change across all scales, i.e., $\forall m \in M(ERS, NRS)(S(m)) \vee \forall m \in M(ERS, NRS)(\neg S(m))$.*

So, a statement that is true with one scale is also true with all other scales, and one that is false with one scale is also false with all other scales. Choosing one scale instead of another basically means adopting a convention, because this choice does not affect the truth value of meaningful statements, like saying that a wooden board object is 23 times as long as another. Instead, suppose we can tell if a software failure is more critical than another on a 5-value criticality measure cr'. For instance, suppose that those five values are $\{1, 2, 3, 4, 5\}$, from 1 (least severe) to 5 (most severe). It is typically meaningless to say that criticality 2 failures are twice as severe as a criticality 1 failure, as the truth value of this statement depends on the specific choice of values. If we choose another scale cr'' with values $\{7, 35, 38, 981, 4365\}$, the truth value of the statement changes, as we would say that the failures in the second category are five times more severe than the ones in the first category. Still, the failures in the second category are given a value that is higher than the value for the first category. The ordering is preserved, and that is where the real information content of the scale lies. This piece of information is preserved by applying a monotonically increasing scale transformation, not just a proportional one. On the contrary, the length of wooden boards cannot undergo any monotonically increasing scale transformation, but only proportional transformations. So, different scales may be subject to different kinds of transformations without any loss of information, which are called admissible transformations.

Definition 7 (Admissible Transformation). *Given a scale (ERS, NRS, m), the transformation of scale f is admissible if $m' = f \circ m$ (i.e., m' is the composition of f and m) and (ERS, NRS, m') is a scale.*

Actually, proportional transformations can be used for a number of different scales (e.g., the typical scales for weight), while monotonically increasing transformations can be used for all sorts of ranking. Measurement Theory identifies five different kinds of scales based on five different kinds of transformations scales can undergo while still preserving their meaningful statements. We now list these different kinds of scales in ascending order of the information they provide.

Nominal Scales. The values of these scales are labels–not necessarily numbers–for categories in which the entities are partitioned, with no notion of order among the categories. Their characterizing invariant property states that the actual labels used do not matter, as long as different labels are used for different categories. Formally, $\forall e_1, e_2 \in E$

$$\forall m \in M(ERS, NRS)(m(e_1) = m(e_2)) \vee \forall m \in M(ERS, NRS)(m(e_1) \neq m(e_2))$$

As the partitioning of the entities into the categories is the information that needs to be preserved, nominal scales can be transformed into other nominal scales via one-to-one transformations.

The programming language in which a program is written is an example of a nominal scale, i.e., we can associate the labels (i.e., values of the scale) C, $Java$, $COBOL$, etc. with each program. As long as programs written in the same language receive the same label and program written in different languages receive different labels, we can adopt programming language names like $alpha$, $beta$, $gamma$, etc.; or $Language1$, $Language2$, $Language3$, etc.; or 1, 2, 3, etc. Note that we do not need to use numbers as values of the measure. Actually, it would be meaningless to carry out even the simplest arithmetic operations. In the Numerical Relational System it is obviously true that "$1 + 2 = 3$," but that would become something nonsensical like $C + Java = COBOL$ just by choosing a different legitimate scale.

As for descriptive statistics, it is well known that the mode (i.e., the most frequent value) is the central tendency indicator that should be used with nominal measures, even though there may be more than one mode in a sample. The arithmetic average cannot be used, as it cannot even be computed, since arithmetic operations are barred. As a dispersion indicator, one may use the Information Content $H(f)$ computed by taking the frequencies of each value as their probabilities, i.e.,

$$H(f) = -\sum_{v \in V} f(v) \log_2 f(v)$$

where $f(v)$ is the frequency of value v.

Association statistical methods can be used too with nominal measures. For instance, suppose that we would like to find a software component that we may want to reuse in your software system and that there are a number of functionally equivalent candidate software components we can choose from and the only information we have is the programming language they are written in. Suppose also that we want to select the software component that has the lowest defect density, but we do not have that piece of information. If defect density data about components are available, association statistical methods like those based on chi-square tests can be used to find out how much we can rely on components written in different languages. So, even though information about defect density is not available and our measure (the programming language) does not involve any number, we can still make an informed and statistically sensible decision.

However, nominal measures only allow the classification of entities into different categories. A nominal measure for the size of program segments could only tell if two segments have the same size or not, but it would not provide any information on whether one program segment is larger or smaller than another one.

Ordinal Scales. In ordinal scales, the entities are partitioned into categories, and the values of these scales are *totally ordered* labels. Their characterizing invariant property states that the actual labels used do not matter, as long as the order of the values that label different categories is preserved. Formally, $\forall e_1, e_2 \in E$

$$\forall m \in M(ERS, NRS)(m(e_1) > m(e_2)) \vee$$
$$\forall m \in M(ERS, NRS)(m(e_1) = m(e_2)) \vee$$
$$\forall m \in M(ERS, NRS)(m(e_1) < m(e_2))$$

As the ordering across the categories is the piece of information that needs to be preserved, ordinal scales can be transformed into other scales via strictly monotonic transformations.

Roughly speaking, ordinal scales are like nominal scales for which, in addition, an ordering on the categories has been defined. Because of the existence of this additional property to be preserved when one scale is transformed into another scale, not all of the possible one-to-one transformations can be used. So, the set of admissible transformations for ordinal scales is a subset of the set of admissible transformations for nominal scales. This reduces the degree of arbitrariness in choosing a scale, and makes an ordinal scale more information-bearing than a nominal scale, because ordinal scales give information about the ordering of the entities and not just their belonging to classes. In our program segment size example, ordinal scales allow us to tell if a program segment is of greater length than another, not just that the two segments have different sizes.

A good example of an ordinal measure is failure criticality, as defined in many bug tracking systems, in which it is possible to associate a criticality value with each bug (for example, in SourceForge, bugs may be a ranked on a nine value scale). Like with nominal scales, it is possible to use numbers as values of ordinal scales, and it is even more tempting than with nominal scales to use arithmetic operations on these numbers. However, this would not be correct, as it would lead to meaningless results. Suppose here that we use an ordinal scale with five values 1, 2, 3, 4, and 5 under the usual numerical ordering. Alternatively, we could have used A, B, C, D, and E, with the usual alphabetical ordering, so A is least severe and E is most severe. While obviously $1+2=3$ in the mere realm of numbers, an operation like that would translate into something like $A+B=C$ if we adopt the alphabetical labels. This statement is meaningful if and only if we can actually say something like "the presence of a bug at criticality A and one at criticality B are equivalent to the presence of a bug at criticality C," under some notion of equivalence. However, this is an additional piece of information, that cannot be inferred from what we know about the mere ordering of the entities in any way. So, this is not a meaningful statement. To have an additional proof of this, let us transform the scale into another numerical scale with values 10, 15, 20, 25, 30. The corresponding statement would become $10+15=20$, which is clearly false. So, transforming a scale into another scale makes the truth value of the statement change, i.e., the statement is not meaningful. The real point here is that we know if one failure is more or less critical than another failure, but we have no idea by how much.

So, one may very well use numbers as values of an ordinal scale, but the kind of mathematical manipulations that can be made must be limited to using $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$. As a consequence it is not allowed to compute the arithmetic average or the standard deviation of a sample of ordinal values.

As for descriptive statistics, the median is the central tendency indicator of choice for ordinal measures. The median of a sample is defined as that value *med* in the sample such that less than half of the data points have values less than *med* and less than half of the data points have values greater than *med*. If the median is not unique, there may be at most two medians in a sample, and they have consecutive values. At any rate, since ordinal scales may be seen as specializations of nominal scales, the descriptive statistics of nominal scales can be applied to ordinal scales too.

Association statistical methods can be used too with ordinal measures. For instance, suppose that we need to find whether failure criticality is statistically related to the effort needed to solve a bug. For instance, suppose we would like to find out if it is true that bugs with higher criticality also take more time to be fixed. Suppose also that this bug fixing effort is measured on an ordinal scale, because the effort collection system allows software engineers to enter values in classes of values like "less than one hour," "between one and four hours," "between four hours and one work day," "between one workday and one work week," and "more than one work week." Statistical indicators are available to investigate this association. For instance, one can use Spearman's ρ or Kendall's τ [19], which provide a measure of the strength of the increasing or decreasing association between two ordinal variables (failure criticality and bug fixing effort, in our example). Also, statistical tests are available to check how statistically significant the associations are. These indicators do not assume that the two variables are linked by any specific functional form (e.g., linear). On the positive side, they can be used to investigate whether there is *any* increasing or decreasing association between two variables. On the negative side, it is not possible to build a specific estimation model, because an estimation model would be based on some functional form that links the two variables. Again, the association statistics for nominal scales can be used with ordinal scales as well.

Summarizing, by using a nominal or an ordinal scale, we can have information about an attribute of a set of entities, but we do not need to use any numbers. The following kinds of scales will require the use of numbers and will provide more refined information about an attribute of a set of entities.

Interval Scales. In interval scales, each entity is associated with a numerical value. Their characterizing invariant property states that the actual values used do not matter, as long as the ratios between all pairs of differences between values are preserved. Formally, by denoting the set of positive real numbers by Re_+, $\forall e_1, e_2, e_3, e_4 \in E$

$$\exists k_1, k_2 \in Re_+, \forall m \in M(ERS, NRS) k_1(m(e_1) - m(e_2)) = k_2(m(e_3) - m(e_4))$$

An interval scale m' can be transformed into another interval scale m'' only via linear transformations $m'' = am' + b$, with $a > 0$, i.e., we can change the origin of the values (by changing b) and the unit of measurement (by changing a). Linear transformations are a subset of strictly monotonic transformations, which are the admissible transformations for ordinal measures. So, again, this reduces the number of possible measures into which an ordinal measure can be transformed, and, again, this makes interval scales even more information-bearing than ordinal scales.

Typical examples of interval scales are calendar time or temperature measured with the scales ordinarily used to this end. For instance, take the Celsius scale for temperatures. It is well known that the origin is conventionally established as the temperature at which water freezes under the pressure of one atmosphere. Also, the 100 Celsius degree mark is conventionally established as the temperature at which water boils under the pressure of one atmosphere. These conventional choices determine the (thereby conventional) extent of one Celsius degree. In addition, it is well known that Celsius degrees can be transformed into, say, Fahrenheit degrees, by means of the following linear transformation relationship

$$Fahrenheit = \frac{9}{5}Celsius + 32$$

It is easy to see that, in addition to the meaningful statements that can be made for ordinal scales, interval scales allow us to make statements in which the ratios of the differences between measurement values are preserved.

Not many software measures are defined at the interval level of measurement. The most important one is probably calendar time, which, for instance, is used during the planning or monitoring of a project. However, the importance of interval scales is that numbers are truly required as measurement values. For one thing, it would not be possible to carry out the linear transformations, otherwise. Some arithmetic manipulations are possible, as shown in the definition. For instance, subtraction between two values of an interval measure provides a result that makes sense. The difference between two dates, e.g., the end and the beginning of a software project, obviously provide the project's duration (which is actually a ratio scale, as we explain in later in this section). Nevertheless, not all possible arithmetic manipulations can be used. It would not make much sense to sum two dates for two events, e.g., May 28, 2005 and July 14, 2007, for instance. Also, if today's Celsius temperature is 20 Celsius degrees, and yesterday's was 10 Celsius degrees, it does not make any sense to say that today is twice as warm as yesterday, as can be easily shown by switching to Fahrenheit temperatures. So, taking the ratio of two interval measure values does not make sense.

Nevertheless, it is meaningful to compute the average value of an interval measure, even though averages are built by summing values if we are interested in comparing two average values. Suppose that the average of the values of a sample is greater than the average of the values of another sample when we use an interval measure. It can be shown that this relationship holds for any other interval scale chosen. So, the average is a good central tendency indicator for interval scales. At any rate, the same holds true for the medians, which can be clearly used for interval scales, which can be seen as a subset of ordinal scales. New dispersion statistics can be added to those "inherited" from ordinal scales, e.g., the standard deviation and the variance. As a matter of fact, they provide a metric evaluation of dispersion, unlike the dispersion indicators of nominal and ordinal scales.

As for association statistics, Pearson's correlation coefficient r [19] can be used when interval scales are involved. However, when using statistical significance tests related to r, it is important to make sure that the assumptions underlying these tests are satisfied. Otherwise, there is a danger of obtaining results that are not statistically valid. At any rate, one can always resort to the association indicators that can be used with ordinal scales. It is true that some statistical power may be lost when using Spearman's ρ or Kendall's τ instead of Pearson's r, but this loss may not be too high. For instance, it has been computed that the so-called Asymptotic Relative Efficiency of Kendall's τ with respect to Pearson's r is 0.912. Roughly speaking, from a practical point of view, this means that 1,000 data points are needed to obtain enough evidence to reach acceptance or rejection of a statistical hypothesis on the association between two interval scales by using Kendall's τ when 912 data points are needed to obtain enough evidence to reach acceptance or rejection of a statistical hypothesis on their correlation. Thus, using Kendall's τ implies having to collect about 8.8% more data points than we would need with Pearson's r. The additional catch is that this value of Asymptotic Relative Efficiency is computed only if the underlying assumptions for using and statistically testing Pearson's r are satisfied. These assumptions may not hold, in practice, so it is usually advisable to use Spearman's ρ and/or Kendall's τ in addition to Pearson's r when carrying out an analysis of the statistical dependence between two interval variables.

At any rate, with interval scales, we can use most of the traditional statistical indicators, because interval scales are truly numerical scales. The next kind of scales removes one of the degrees of arbitrariness intrinsic to interval scales: the origin is no longer conventional.

Ratio Scales. Each entity is associated with a numerical value by ratio scales. Their characterizing invariant property states that the actual values used do not matter, as long as the ratios between all the pairs of values are preserved. Formally, $\forall e_1, e_2 \in E$

$$\exists k_1, k_2 \in Re_+, \forall m \in M(ERS, NRS) k_1 m(e_1) = k_2 m(e_2)$$

So, this property implies that a ratio scale m' can be transformed into another ratio scale m'' only via proportional transformations $m'' = am'$, with $a > 0$. This shows that it is possible to change the measurement unit by changing a, but not the origin as the value 0 in one scale correspond to the value 0 in all other scales, so it is invariant. The above formula shows that the set of admissible transformations for ratio scales is a subset of the admissible transformations for interval scales, the difference between ratio scales and interval scales basically being that for ratio scales have a natural origin, which is invariant, while a conventional origin can be chosen for interval scales. Ratio scales obviously can only take numerical values, like interval scales.

Size (e.g., volume or mass) is typically represented by ratio scales, for instance. Time durations or temperature intervals may be represented with ratio scales and, in general, the difference between two values of an interval measure is a ratio scale. In Software Engineering Measurement, software size is typically represented via ratio scales and so is development effort.

Legitimate operations involving ratio scales include differences, ratios, and sums. For instance, the size of a program segment composed of two program segments may be obtained as the sum of the sizes of those two program segments. As for descriptive and association statistics, there is basically the geometric mean that can be used with ratio scales, in addition to the other descriptive and association statistics that can also be used with interval scales. From a practical point of view, this shows that there is a real divide between ordinal and interval measures. The former are nonnumeric, while the latter are numerical ones.

Absolute Scales. Absolute scales are the most "extreme" kind of measures, in a sense. Each entity is associated with a numerical value in absolute scales, and their invariant property states that the actual values used *do* matter, since the only admissible transformation is identity, i.e., an absolute scale cannot be transformed into anything other than itself. Formally, $|M(ERS, NRS)| = 1$.

Again, this transformation is a subset of the possible transformations of ratio scales. The measurement unit is fixed and cannot be chosen conventionally. So, these scales are the most informative ones, since their values bear information themselves, and not only in relationship.

Statistics of Scales. The description of scale types is not simply a theoretical exercise, but it has important practical consequences, as we have already discussed. Some mathematical operations may not be applied to measures of certain measurement levels, e.g., summing may not be used for the numerical values of nominal, ordinal, or even interval measures. Based on the scale type of a measure, different indicators of central tendency can be used without resulting in meaningless statements, i.e., the mode for nominal scales, the median as well for ordinal scales, the arithmetic mean as well for interval scales, and the geometric mean as well for ratio and absolute scales. the same applies to dispersion indicators and statistical tests. Table 1 summarizes a few well-known indicators of central tendency, dispersion, and association that are appropriate for each scale type. In each cell of columns "Central Tendency," "Dispersion," and "Association," we report only the indicators that can be used for scales that are at least on that measurement level. So, these indicators can be used for scales at higher measurement levels. For instance, as already noted, the median can be used for ordinal, interval, ratio, and absolute scales, but not for nominal scales.

3.2 Additional Issues on Scales

Two additional issues on scales are often given some more attention, in practical use and in theoretical debates. We briefly discuss them here.

Table 1. Characteristics of different scale types

Scale Type	Admissible Transformation	Examples	Central Tendency	Dispersion	Association
Nominal	Bijections	Gender, Progr. Language	Mode	Information Content	Chi-square
Ordinal	Monotonically increasing	Preference, Fail. Criticality	Median	Interquartile range	Spearman's ρ, Kendall's τ
Interval	Linear	Temperature, Milestone Date	Arithmetic Mean	Standard Deviation	Pearson's r
Ratio	Proportional	Mass, Software Size	Geometric Mean		
Absolute	Identity	Probability			

Subjective Scales. An "objective" measure is one for which there is an unambiguous measurement procedure, so it is totally repeatable. A "subjective" measure is computed via a measurement procedure that leaves room for interpretation on how to measure it, so different people may come up with different measurement values for the same entity, the same attribute, and the same measure itself. It is usually believed that objective measures are always better than subjective measures, but this claim needs to be examined a bit further.

- For some attributes, no objective measure exists. The number of faults in a software program cannot be measured, so we may resort to subjective evaluations for it.
- Even when it is theoretically possible to use an objective measure, it may not be practically or economically viable to use that measure. For instance, the number of faults in a software application with a finite input domain may be measured by executing the application with all possible input values, but this would be impractical.
- Some measures look more objective than they actually are. Take this objective measure of reliability for a software application a: $objReliability(a) = 0$ if a has at least one failure in the first month of operation, and otherwise $objReliability(a) = 1$. Some failures may occur in the first month of operation, but they may go unnoticed, or their effect may surface several months later.
- An objective measure may be less useful than a subjective measure anyway. Take two measures (a subjective and an objective one) for two different attributes. Nothing guarantees that the objective measure is more useful than the subjective one to predict some variable of interest. Even for the same attribute, a subjective measure may be more useful than an objective one, if it captures that attribute more sensibly.

Indirect Scales. It is commonly said that measures built by combining other measures are indirect ones. For instance, fault density represented as the ratio between the number of uncovered faults and LOC (the number of lines of code of a program segment) would be an indirect measure. However, even among measurement theoreticians, there is no widespread consensus that it is actually necessary or useful to make the distinction between direct and indirect measures, or that it is even possible to make this distinction. One of the points is that even indirect scales should satisfy exactly the same requirements as direct scales, since indirect scales are scales anyway, so they should be built by using an Empirical Relational System, a Numerical Relational System, a function between them, and a Representation Condition. If all of these theoretical definition elements are in place, then there is no reason to distinguish between direct and indirect scales anyway [34].

3.3 Evaluation of Measurement Theory

Measurement Theory is the reference, ideal model to which one should tend in the definition of a measure. A measure defined in such a way as to comply with the Representation Condition is a legitimate measure for an attribute. However, Measurement Theory's constraints may be too strict. For instance, LOC does not comply with the Measurement Theory's requirements for size measures. Overall, Measurement Theory has been used to eliminate measures that have been proposed for the quantification of software attributes, but it has not been helpful or productive when it comes to defining new measures. Also, other than the modeling of size, no other use of Measurement Theory is known in Software Measurement. Thus, especially in this phase, in which Software Measurement has not reached a sufficient degree of maturity, it is useful to use other approaches like Axiomatic Approaches (see Sections 4 and 5), which have more relaxed requirements and so they do not eliminate a number of measures that Measurement Theory would reject. Measures may therefore get a chance to be better refined later on, and this contributes to having a better understanding of the characteristics of software attributes too.

4 Axiomatic Approaches

Other approaches have been used to represent the properties that can be expected of software attributes, e.g., [33,39,21,10,25,31,32,23]. The underlying idea has been long used in mathematics to define concepts via sets of axioms. The axioms for distance are a very well-known example. The distance d between two elements x and y of any set S is defined as a real-valued function $d : S \times S \to Re$ that satisfies the following three axioms.

Distance Axiom 1 Nonnegativity. *The distance between any two elements is nonnegative, i.e.,* $\forall x, y \in S(d(x, y) \leq 0)$*, and it is zero if and only if the two elements coincide, i.e.,* $\forall x, y \in S(d(x, y) = 0 \Leftrightarrow x = y)$*.*

Distance Axiom 2 Symmetry. *The distance between any two elements x and y is the same as the distance between y and x, i.e.,* $\forall x, y \in S(d(x, y) = d(y, x))$*.*

Distance Axiom 3 Triangular Inequality. *Given three elements, the sum of the distances between any two pairs is greater than the distance between the other pair of elements, i.e.,* $\forall x, y, z \in S(d(x, y) + d(y, z) \geq d(x, z))$.

These axioms have been applied to very concrete sets, such as sets of points in the physical world, and much more abstract ones, such as sets of functions in mathematics. Different functions that satisfy these axioms can be defined, even within the same application domain. The choice of a specific distance measure depends on a number of factors, including the measurement goals, tools, and resources. No matter the specific application, these three axioms are commonly accepted as the right axioms that capture what a distance measure should look like, so they are no longer a topic for debates, since a broad consensus has been reached about them. Other sets of axioms have been defined for other attributes (e.g., the Information Content $H(p)$ of a discrete probability distribution p, which is the basis of Information Theory).

The set of axioms for distance functions is certainly no longer controversial and its introduction is based on the properties of distances between physical points. The set of axioms for Information Content are quite recent and they address a more abstract attribute, but, there is now a widespread consensus about them. The introduction of Axiomatic Approaches in Software Measurement, is even more recent, due to the novelty of Software Engineering and, more specifically, of Software Measurement, which, as already noted, deals with somewhat abstract attributes of intangible entities. Therefore, it is natural that there has not been enough time to reach a broad consensus around specific sets of axioms for software attributes. Nevertheless, one of the main advantages of using an axiomatic approach over using an "operational" approach, i.e., providing a measure as if it was an "operational" definition for the attribute, is that the expected properties of the measures of the attribute are clearly spelled out. Thus, a common understanding of the properties can be reached and disagreements can focus on specific properties instead of more vaguely defined ideas. At any rate, Axiomatic Approaches have already been used to check if existing measures satisfy a specific set of axioms for the attribute that they are supposed to measure. Perhaps more importantly, these approaches have been used as guidelines during the definition of new measures.

Also, it must understood that these Axiomatic Approaches do not have the same "power" as Measurement Theory. Rather, the set of axioms associated with a specific attribute (e.g., software size) should be taken as sets of necessary properties that need to be satisfied by a measure for that software attribute, but not sufficient ones. Thus, those measures that do not satisfy the set of axioms for a software attribute cannot be taken as legitimate measures for that attribute. The measures that do satisfy the set of axioms are candidate measures for that software attribute, but they still need to be better examined. Finally, like with Measurement Theory, the measures that comply with the theoretical validation still need to undergo a thorough empirical validation that supports their practical usefulness. We address this issue in Section 7.

4.1 Weyuker's Complexity Axioms

Weyuker's approach [39] represents one of the first attempts to use axioms, to formalize the concept of program complexity. The approach introduces a set of nine axioms, which we number $W1, \ldots, W9$. Weyuker's approach was defined for the complexity of so called "program bodies," which we have called program segments so far. So, the approach was defined for the complexity of sequential programs or subroutines. The composition of program segments is concatenation and it is denoted by ';':$ps_1; ps_2$ denotes the concatenation of two program segments ps_1 and ps_2.

W1. A complexity measure must not be "too coarse" (part 1)

$$\exists ps_1, ps_2 (Complexity(ps_1) \neq Complexity(ps_2))$$

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

W3. A complexity measure must not be "too fine." There exist distinct program segments with different complexity

$$\exists ps_1, ps_2 (Complexity(ps_1) = Complexity(ps_2))$$

W4. Functionality and complexity have no one-to-one correspondence between them

$$\exists ps_1, ps_2 (ps_1 functionally\_equivalent\_tops_2) \wedge \\ (Complexity(ps_1) \neq Complexity(ps_2))$$

W5. Concatenating a program segment with another program segment may not decrease complexity

$$\forall ps_1, ps_2 (Complexity(ps_1) \leq Complexity(ps_1; ps_2)) \wedge \\ (Complexity(ps_2) \leq Complexity(ps_1; ps_2))$$

W6. The contribution of a program segment in terms of the overall program may depend on the rest of the program

$$\exists ps_1, ps_2, ps_3 (Complexity(ps_1) = Complexity(ps_2)) \wedge \\ (Complexity(ps_1; ps_3) \neq Complexity(ps_2; ps_3))$$
$$\exists ps_1, ps_2, ps_3 (Complexity(ps_1) = Complexity(ps_2)) \wedge \\ (Complexity(ps_3; ps_1) \neq Complexity(ps_3; ps_2))$$

W7. A complexity measure is sensitive to the permutation of statements. There exist ps_1 and ps_2, such that ps_1 is obtained via a permutation of the statements of ps_2 and $Complexity(ps_1) \neq Complexity(ps_1)$.

W8. A complexity measure is not sensitive to the specific identifiers used. If ps_1 is obtained by renaming the identifiers of ps_2, then

$$Complexity(ps_1) = Complexity(ps_2)$$

W9. There are program segments whose composition has a higher complexity than the sum of their complexities

$$\exists ps_1, ps_2 (Complexity(ps_1) + Complexity(ps_2) < Complexity(ps_1; ps_2))$$

The following analysis of Weyuker's axioms may shed some light on their characteristics and the kind of complexity that they are meant to describe.

– Axioms $W1$, $W2$, $W3$, $W4$, $W8$ do not characterize complexity alone, but they may be applied to all syntactically-based product measures, e.g., size measures. At any rate, they need to be made explicit in an axiomatic approach.
– Axiom $W5$ is a monotonicity axiom which shows that Weyuker's axioms are about "structural" complexity and not "psychological" complexity. Suppose that program segment ps_1 is an incomplete program, and the complete program is actually given by the concatenation $ps_1; ps_2$. It may very well be the case that the entire program is more understandable than ps_1 or ps_2 taken in isolation, as some coding decisions may be easier to understand if the entire code is available.
– Axiom $W7$ shows that the order of the statements does influence complexity. Without this axiom, it would be possible to define a control-flow complexity measure that is totally insensitive to the real control flow itself, as the statements in a program segment could be arbitrarily rearranged without affecting the value of a control-flow complexity measure.
– Axiom $W8$ too shows that Weyuker's axioms are about "structural" complexity, not "psychological" complexity. Renaming does not have any impact on Weyuker's concept of complexity, but it is obvious that, if a program segment's variables were renamed by using meaningless, absurd, or misleading names, the program segment's understandability would be certainly heavily affected, and, in turn, its "psychological" complexity.
– Axiom $W9$ is probably the one that most characterizes complexity, even if it does not come in a "strong" form, since it uses an existential quantification. The idea, however, is that there are cases in which the complexity of a program segment is higher than the sum of the complexities of its constituent program subsegments. This axiom, however, does not rule out the existence of two program segments whose composition has a *lower* complexity than the sum of their complexities

$$\exists ps_1, ps_2 Complexity(ps_1) + Complexity(ps_2) > Complexity(ps_1; ps_2)$$

5 A Unified Axiomatic Approach for Internal Software Attributes

We now illustrate the proposal initially defined by Briand, Morasca, and Basili [10,25] and its later refinements by Morasca [23]. This proposal addresses several different software product attributes, including size, complexity, cohesion, and coupling, which we discuss in this section. Based on an abstract graph-theoretic model of a software artifact description of a software artifact, each software attribute is associated with a set of axioms that its measures should satisfy. Thus, unlike in other approaches, a set of different software attributes are studied in a unified framework that makes it easier to identify the similarities and differences between software attributes. In addition, as it is based on an abstract graph-theoretic representation, this axiomatic approach can be applied for measures of many different artifacts that are encountered during the software life cycle, and not just software code.

5.1 Systems and Modules

The basic idea is that a system is a multigraph, where each arc is associated with a multiset of relationships, and each relationship has a type.

Definition 8. *System. A system S is a pair $S =< E, R >$, where*

- *E represents the set of elements of S*
- *$R \in N^{E \times E \times T}$*

where T is a finite set of types of relationships (N is the set of natural numbers, including 0).

The idea is that a software artifact contains a set basic elements, which are represented as the nodes of the multigraph. These elements are connected by possibly more than one relationship of possibly different types. The relationships between the elements are therefore represented by the multisets of typed arcs.

As an example, take the class diagram in Fig. 1, built by using a UML-like notation in which classes (like C or D) may belong to two packages, so this notation is even more general than standard UML. The classes are the elements of the system. The arcs are annotated with different types, e.g., aggregations, inheritance, use, etc., and two classes may very well be connected by several relationships, of the same or of different types (see classes K and L). In addition, a UML-like diagram may not even represent all of the relationships existing between classes. For instance, inheritance is a transitive relation, and transitive relationships are not explicitly represented. In Fig. 2, the aggregation between M and Q gets inherited by N, O, and P. So, the actual set of relationships may be greater than those that are explicitly mentioned in the graph.

To define axioms for internal software attributes defined for software artifacts, we first need to define an "algebra" whose operations are introduced next. In what follows, the same symbol (e.g., $\cup$ for union) may denote an operation between

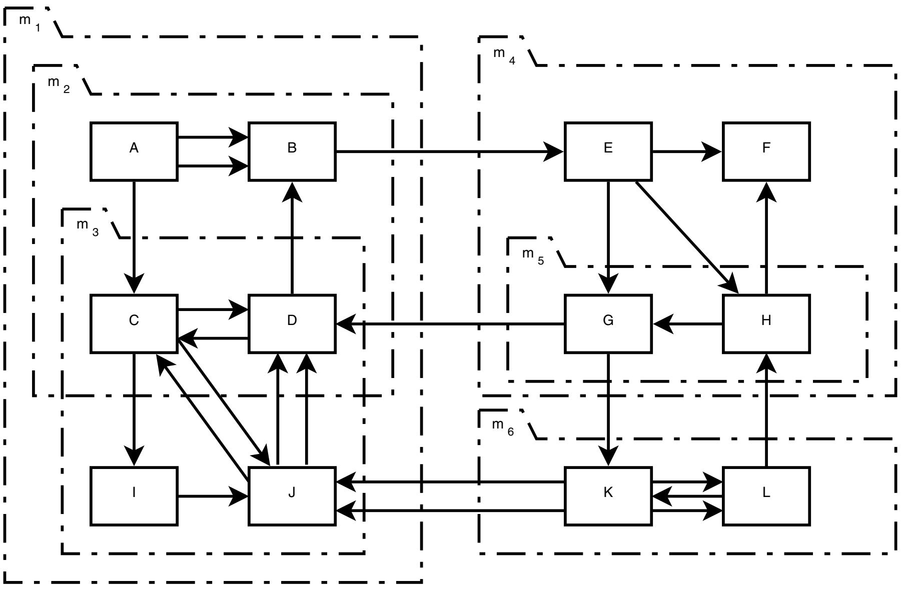

Fig. 1. Representation of a system and its modules in a UML-like language

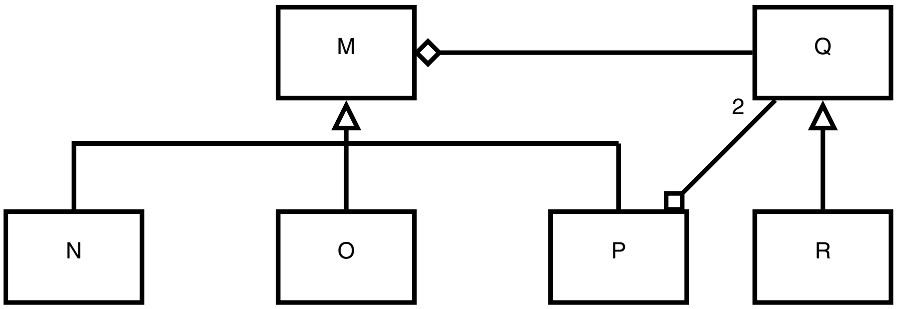

Fig. 2. A UML-like Class Diagram

– sets when sets of elements are involved
– multisets when multisets of typed relationships are involved
– modules (see Definition 9) when modules are involved.

These operations are different, but no confusion will arise because they never involve operands of different nature. For instance, no union will be defined between a multiset of typed relationships and a module.

For completeness, we here provide the meaning of these operations between two typed multisets of relationships R_1, R_2.

Inclusion. $R_1 \subseteq R_2 \Leftrightarrow \forall << a, b, t >, n_1 > \in R_1$,
$\exists << a, b, t >, n_2 > \in R_2 \wedge n_1 \leq n_2$, i.e., R_2 contains at least all the occurrences of the typed relationships in R_1.

Union. $R_3 = R_1 \cup R_2 \Leftrightarrow \forall << a, b, t >, n_3 > \in R_3$,
$\exists << a, b, t >, n_1 > \in R_1, << a, b, t >, n_2 > \in R_2, n_3 = n_1 + n_2$, i.e., R_3 gathers all the occurrences of the typed relationships in R_1 and R_2.

Intersection. $R_3 = R_1 \cap R_2 \Leftrightarrow \forall << a, b, t >, n_3 > \in R_3, \exists << a, b, t >, n_1 > \in R_1, << a, b, t >, n_2 > \in R_2, n_3 = min\{n_1, n_2\}$, i.e., R_3 contains all the occurrences of typed relationships in common to R_1 and R_2.

Using operations like the union implies that parts of a system be identifiable so they can be put together. Also, some internal software attributes naturally require that parts of a system be identifiable. For instance, coupling is typically defined as an attribute defined for the cooperating parts of a software system, or for the entire system. These parts of a system are actually subsystems, which we call *modules.*

Definition 9. *Module. Given a system* $S = < E, R >$*, a module* $m = < E_m, R_m >$ *is a system such that* $E_m \subseteq E \wedge R_m \subseteq R$.

For maximum generality and simplicity, a module is simply a subsystem, with no additional characteristics (e.g., an interface). At any rate, a module m of a system will contain a multiset of relationships of its own, and there will be a (possibly empty) multiset of relationships that link m to the rest of the system, which will be denoted as $OuterR(m)$. In Fig. 1, UML-like packages m_1, m_2, m_3, m_4, m_5, m_6, may be interpreted as modules. It will be our convention in the remainder of the paper that the set of elements and the multiset of relationships of a system or a module have the same subscript as the system or module, unless otherwise explicitly specified (e.g. $m_1 = < E_1, R_1 >$).

We can now introduce a few operations and definitions that compose the "algebra" of modules upon which the sets of axioms will be defined.

Inclusion. Module m_1 is said to be included in module m_2 (notation: $m_1 \subseteq m_2$) if $E_1 \subseteq E_2 \wedge R_1 \subseteq R_2$. In Fig. 1, $m_5 \subseteq m_4$.

Union. The union of modules m_1 and m_2 (notation: $m_1 \cup m_2$) is the module $< E_1 \cup E_2, R_1 \cup R_2 >$. In Fig. 1, $m_1 = m_2 \cup m_3$.

Intersection. The intersection of modules m_1 and m_2 (notation: $m_1 \cap m_2$) is the module $< E_1 \cap E_2, R_1 \cap R_2 >$. In Fig. 1, $m_2 \cap m_3$ is the module whose elements are classes C and D and whose relationships are $<< C, D, t >, 1 >$ and $<< D, C, u >, 1 >$ (assuming that they have type t and u, respectively).

Empty Module. Module $< \oslash, \oslash >$ (denoted by $\oslash$) is the empty module.

Disjoint Modules. Modules m_1 and m_2 are said to be disjoint if $m_1 \cap m_2 = \oslash$. In Fig. 1, m_3 and m_6 are disjoint.

Unconnected Modules. Two disjoint modules m_1 and m_2 of a system are said to be unconnected if $OuterR(m_1) \cap OuterR(m_2) = \oslash$. In Fig. 1, m_4 and m_6 are unconnected, while m_3 and m_6 are not unconnected.

5.2 Axiom Sets and Derived Properties

We here introduce a set of axioms for a few internal software attributes of interest. In addition, we show properties that can be derived as implications of those axioms, to further check whether the modeling of an internal software attributes is consistent with the intuition on it. As a matter of fact, the decision as to which properties are more basic and should be taken as axioms and which are derived properties is somewhat subjective. We mostly take properties satisfied by ratio measures as the axioms and, often, properties satisfied by ordinal measures are derived. (Each axiom and property is annotated by the level of measurement of the measures to which the axiom or property can be applied to.) The derived properties are "weaker" than the axioms base and are often satisfied by measures that are ordinal or nominal and not necessarily ratio ones. This is not just a theoretical exercise, but can guide the building of ordinal or nominal measures, instead of only ratio ones.

Size. The idea underlying the first axiom is that the size of a module composed of two possibly overlapping modules is not greater than the sum of the sizes of the two modules by themselves.

Size Axiom 1 Union of Modules (ratio scales). *The size of a system S is not greater than the sum of the sizes of two of its modules m_1 and m_2 such that each element of S is an element of either m_1 or m_2 or both*

$$E = E_1 \cup E_2 \Rightarrow Size(S) \leq Size(m_1) + Size(m_2)$$

For instance, $Size(m_1) \leq Size(m_2) + Size(m_3)$ in Fig. 1.

However, when the two modules are disjoint, size is additive.

Size Axiom 2 Module Additivity (ratio scales). *The size of a system S is equal to the sum of the sizes of two of its modules m_1 and m_2 such that any element of S is an element of either m_1 or m_2 but not both*

$$E = E_1 \cup E_2 \wedge E_1 \cap E_2 = \oslash \Rightarrow$$
$$Size(S) = Size(m_1) + Size(m_2)$$

Thus, $Size(m_1 \cup m_6) = Size(m_1) + Size(m_6)$ in Fig. 1.

A number of properties can be derived from these two base axioms, as follows:

– the size of the empty system is zero (ratio scales);

- the size of a system is nonnegative (ratio scales);
- the size of a system is not lower than the size of the empty system; though it can be clearly inferred from the first two derived properties, this is a property that can be used for ordinal scales too (ordinal scales);
- adding elements to a system cannot decrease its size (ordinal scales);
- relationships have no impact on size, i.e., two systems with the same elements will have the same size (nominal scales);
- a measure of size is computed as the sum of the "sizes" of its elements: if we take each element e of a system and we build a module that only contains e, then compute the size of this newly defined module, and then sum the sizes of all these newly defined modules, we obtain the value of the size of the entire system (ratio scales).

The last two derived properties thus show that size is based on the elements of a software system and not on its relationships.

It turns out that this axiomatic definition of size is closely related to the axiomatic definition of what is known as "measure" in Measure Theory [30], an important branch of Mathematics that is a part of the basis of the theory of differentiation and integration in Calculus. So, this places these axioms on even firmer mathematical grounds.

Examples of size measures according to this axiomatic approach: $\#Statements$, LOC, $\#Modules$, $\#Procedures$, Halstead's $Length$ [17], $\#Unique\_Operators$, $\#Unique\_Operands$, $\#Occurrences\_of\_Operators$, $\#Occurrences\_of\_Operands$, WMC [13]. Instead, these are not size measures: Halstead's $Estimator\_of\_length$ and $Volume$ [17].

Complexity. We are dealing here with internal software attributes, so we here mean "structural" complexity, and not some kind "psychological" complexity, which would be an external software attribute. Complexity is based on the relationships among system elements, unlike size.

The idea underlying the first axiom, which characterizes complexity the most, is that the complexity of a system is never lower than the sum of the complexities of its modules taken in "isolation," i.e., when they have no relationships in common, even though they may have elements in common.

Complexity Axiom 1. Module Composition (ratio scales). *The complexity of a system S is not lower than the sum of the complexities of any two of its modules m_1, m_2 with no relationships in common*

$$S \supseteq m_1 \cup m_2 \wedge R_1 \cap R_2 = \emptyset \Rightarrow$$
$$Complexity(S) \geq Complexity(m_1) + Complexity(m_2)$$

Suppose that the two modules m_1 and m_2 in Complexity Axiom 1 have elements in common. All of the transitive relationships that exist in m_1 and m_2 when they are taken in isolation still exist in S. In addition, S may contain new transitive relationships between the elements of m_1 and m_2, which do not exist in either module in isolation. So, the complexity of S is not lower than the sum of the complexities of the two modules in isolation. For instance, in Fig. 1, $Complexity(m_1) \geq Complexity(m_2) + Complexity(m_3)$.

When a system is made up of two unconnected modules, complexity is additive

Complexity Axiom 2. Unconnected Module Additivity (ratio scales). *The complexity of a system S composed of two unconnected modules m_1, m_2 is equal to the sum of the complexities of the two modules*

$$\begin{aligned} S = m_1 \cup m_2 \wedge \\ m_1 \cap m_2 = \oslash \wedge OuterR(m_1) \cap OuterR(m_2) = \oslash \Rightarrow \\ Complexity(S) = Complexity(m_1) + Complexity(m_2) \end{aligned}$$

We now describe a few derived properties for complexity:

- a system with no relationships has zero complexity (ratio scales);
- the complexity of a system is nonnegative (ratio scales);
- the complexity of a system is not lower than the complexity of a system with no relationship, which can be clearly inferred from the first two derived properties; however, this is a property that can be used for ordinal scales too (ordinal scales);
- adding relationships to a system cannot decrease its complexity (ordinal scales);
- elements have no impact on complexity, i.e., two systems with the same relationships will have thee same complexity (nominal scales).

Summarizing, as opposed to size, complexity depends on relationships and not on elements.

These measures may be classified as complexity measures, according to the above axioms: Oviedo's data flow complexity measure DF [29], $v(G) - p$, where $v(G)$ is McCabe's cyclomatic number and p is the number of connected components in a control-flow graph [22]. These measures do not satisfy the above axioms: Henry and Kafura's information flow complexity measure [18], *RFC* and *LCOM* [13].

Cohesion. Cohesion is related to the *degree* and not the *extent* with which the elements of a module are tied to each other. Thus, cohesion measures are normalized.

Cohesion Axiom 1. Upper Bound (ordinal scales). *The cohesion of a module m is not greater than a specified value Max, i.e., $Cohesion(m) \leq Max$.*

Elements are linked to each other via relationships, so adding relationships does not decrease cohesion.

Cohesion Axiom 2. Monotonicity (ordinal scales). *Let modules* $m_1 =< E, R_1 >$, $m_2 =< E, R_2 >$ *be two modules with the same set of elements* E, *and let* $R_1 \subseteq R_2$. *Then,* $Cohesion(m_1) \leq Cohesion(m_2)$.

A module has high cohesion if its elements are highly connected to each other. So, if we put together two modules haphazardly and these two modules are not connected to each other, we cannot hope that the cohesion of the new module will be greater than the cohesion of each the two original modules separately.

Cohesion Axiom 3. Unconnected Modules (ordinal scales). *Let* m_1 *and* m_2 *be two unconnected modules, then,*

$$max\{Cohesion(m_1), Cohesion(m_2)\} \geq Cohesion(m_1 \cup m_2)$$

Thus, in Fig. 1, we have

$$Cohesion(m_4 \cup m_6) \leq max\{Cohesion(m_4), Cohesion(m_6)\}$$

As a side note, these first three axioms may be safely applied to ordinal measures (e.g., a measure like Yourdon and Constantine's [40]).

The following axiom may be satisfied only by ratio measures.

Cohesion Axiom 4. Null Value (ratio scales). *The cohesion of a module with no relationships* $m =< E, \oslash >$ *is null, i.e.,* $Cohesion(m) = 0$.

The above axioms imply the following property:

– the cohesion of a module is not lower than the cohesion of a module with no relationships (ordinal scales).

Examples of cohesion measures according to the above axioms: $PRCI$, $NRCI$, $ORCI$ [11].

Coupling. As opposed to cohesion, the coupling of a module in a system is related to the *amount* of connection between the elements of a module and the elements of the rest of the system. The interconnections may be direct or transitive. So, adding a relationship, whether internal to the module or belonging to its set of outer relationships, can never decrease coupling.

Coupling Axiom 1. Monotonicity (ordinal scales).
Adding a new relationship to a module m_1 *or to its set of outer relationships* $OuterR(m_1)$ *does not decrease its coupling. So, if* m_2 *is a module such that* $E_2 = E_1$, *we have*

$$OuterR(m_2) \supseteq OuterR(m_1) \wedge R_2 \supseteq R_1 \Rightarrow \\ Coupling(m_2) \geq Coupling(m_1)$$

At any rate, if a module has no outer relationships, its elements are not connected with the rest of the system, so its coupling is zero.

Coupling Axiom 2. Null Value (ratio scales). *The coupling of a module with no outer relationships is null.*

Suppose now that we take two modules and put them together. The relationships from one to the other used to be outer ones, but become internal ones after the merging. Thus, we have lost some couplings of the two initial modules in the new module, whose coupling is not higher than the sum of the couplings of the two modules. In Fig. 1, when modules m_2 and m_3 are merged into module m_1 the relationships to and from m_4, m_5, and m_6 are still outer relationships for m_1, but the relationships between m_2 and m_3 have become internal relationships for m_1 (so, they may also contribute to the cohesion of m_1).

Coupling Axiom 3. Merging of Modules (ratio scales). *The coupling of the union of two modules m_1, m_2 is not greater than the sum of the couplings of the two modules*

$$Coupling(m_1 \cup m_2) \leq Coupling(m_1) + Coupling(m_2)$$

However, if the two original modules that got merged were not connected, no coupling has been lost and the new modules has exactly the same coupling as the two original modules.

Coupling Axiom 4. Unconnected Modules (ratio scales). *The coupling of the union of two unconnected modules is equal to the sum of their couplings*

$$m_1 \cap m_2 = \oslash \wedge OuterR(m_1) \cap OuterR(m_2) = \oslash \Rightarrow$$
$$Coupling(m_1 \cup m_2) = Coupling(m_1) + Coupling(m_2)$$

So, $Coupling(m_4 \cup m_6) = Coupling(m_4) + Coupling(m_6)$ in Fig. 1.

Like with the other internal attributes, derived properties can be found, as follows:

- the coupling of a module is nonnegative (ratio scales);
- the coupling of a module is not less than the coupling of a module with no outer relationships (ordinal scales).

Among the measures that may be classified as coupling measures according to the above axioms are: TIC and DIC [11], CBO and RFC [13]. Fenton's coupling measure [16] does not satisfy the above axioms.

5.3 Relationships between Software Attributes

Table 2 summarizes the main characteristics of software attributes for a module m of a system according to the unified axiomatic approach described in this section. We report 1) the condition for the attribute to assume value zero in column "Null Value," 2) the variable with respect to which the attribute has a monotonic behavior in column "Monotonicity," and 3) the condition for additivity in column "Additivity," if any.

One of the goals of this axiomatic approach is to identify similarities, differences, and relationships between attributes, as we now concisely discuss.

Table 2. Characteristics of different software attributes

Attribute	Null Value	Monotonicity	Additivity
Size	$E_m = \emptyset$	E_m	Separate modules
Complexity	$R_m = \emptyset$	R_m	Unconnected modules
Cohesion	$R_m = \emptyset$	R_m	NO
Coupling	$OuterR(m) = \emptyset$	$OuterR(m) \cup R_m$	Unconnected modules

Size vs. Complexity These are the main differences in the properties of size and complexity

- size is based on elements, complexity is based on relationships
- the inequalities about the sums of sizes and complexities in Size Axiom 1 and Complexity Axiom 1 go in opposite directions
- complexity cannot be interpreted as the amount of relationships, as if it was the “size” of the set of relationships, while size is the sum of the “sizes” of the individual elements.

On the other hand, both size and complexity have additivity properties, though under different conditions (see Size Axiom 2 and Complexity Axiom 2).

Complexity vs. Cohesion Complexity and cohesion of a module share a number of similarities as both

- depend on the relationships within the module
- are null when there are no relationships in the module
- increase when a relationship is added to the relationships of the module.

It is possible to show that cohesion measures can actually be defined as absolute measures as follows. Given a complexity measure cx, for any given module m, suppose that there exists $cx_M(m)$ a maximum possible value for cx when it is applied to the elements of module m. This may be reasonable, as there is a finite number of elements in m, and the elements may be linked by a limited number of relationships. Then, $ch(m) = cx(m)/cx_M(m)$ is a cohesion measure. This has two important consequences.

1. From a practical point of view, cohesion may increase when complexity increases. This might explain why sometimes cohesion measures are not very well related to fault-proneness [8], as the positive effect of the increase in cohesion on error-proneness is somewhat masked by the negative effect of an increase in complexity.
2. From a theoretical point of view, an equation like $ch(m) = cx(m)/cx_M(m)$ may used as a starting point to find quantitative relationships among attributes, as is usual in many scientific disciplines.

Complexity vs. Coupling. Both complexity and coupling of a module

- are null when there are no relationships in the module and outside it
- increase when a relationship is added to the relationships of the module.

One characterizing difference between complexity and coupling is that, when merging two disjoint modules are merged in a module, the complexity of the resulting module is not less than the sum of the complexities of the original modules, while the coupling of the resulting module is not greater than the sum of the couplings of the original modules.

6 External Software Attributes: Probability Representations

As explained in Section 2, a number of different external software attributes are of interest for several categories of software "users," depending on their specific goals and the type of application at hand. For instance, usability may be very important for the final users of web applications, while time efficiency may be a fundamental external software attribute for the users of a real-time system, which must deliver correct results within a specified time interval. As for practitioners, every decision made during software development is made, implicitly or explicitly, based on some external software attribute. For instance, when a decision is made between two alternative designs, a number of external software attributes are implicitly or explicitly taken into account, e.g., maintainability, portability, efficiency.

External software attributes may be conflicting. Increasing one may negatively affect others, so a satisfactory trade-off must be reached among them. Being able to assess these qualities may provide users and practitioners with a way to base decisions on firmer grounds and evaluate whether a software product's or component's quality is satisfactory according to a user's or practitioner's goals, and identify a product's or component's strengths and weaknesses.

A number of proposals have appeared in the literature to quantify these external software attributes (e.g., among several others, maintainability [28], usability [35]). In addition, standards have been defined to define the qualities (i.e., external software attributes) of software products, and, more generally, software artifacts. For instance, the ISO9126 standard [1,2] defines quality by means of 6 characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. These characteristics, in turn, are defined in terms of subcharacteristics in a tree-like structure, and measures have been proposed for them too. (An additional characteristic is called quality in use, to summarize the quality as perceived by the user.)

Standards like ISO9126 are useful as reference frameworks, but they may turn out to be too general, as they are meant to address the development of many different kinds of software. So, they do not base the definition and quantification of software qualities on precise, formal, and unambiguous terms, which is what one would expect from measurement activities, which are among the most precise, formal, and unambiguous activities in engineering and scientific disciplines. It is probably impossible to remove all subjectivity and uncertainty in Empirical Software Engineering, due to the number of different factors that influence software production, and especially its being so heavily human-intensive. However, because of the nature of Software Engineering, it is important that the degree of subjectivity and uncertainty be reduced, and, most of all, formalized. Thus, external software attributes should be based on firm, mathematical grounds, to remove subjectivity and uncertainty to the extent possible, and highlight their possible sources and the factors that may influence them. Theoretically sound and sensible ways to measure external software attributes will help prevent the quantification of external software attributes via ill-defined measures or not fully justified approaches.

In this section, we describe a unified probability-based framework for measuring external software attributes [24], which shows that external software attributes should be quantified by means of probabilistic *estimation models* instead of *measures* as defined in Section 3.1.

We discuss the problems associated with using measures to quantify external software attributes (Section 6.1) and then describe of so-called "probability representations" along with their advantages (Section 6.2). Probability Representations are a part of Measurement Theory that is often neglected in Software Measurement, even though they have already been implicitly used in Software Measurement in the modeling and quantification of software reliability [27], for instance. Software reliability can be viewed as a "success story" in the modeling of external software attributes. We describe how it is possible to put another important external software attribute, i.e., software modifiability, on firm mathematical grounds (Section 6.3), to show another useful application of Probability Representations. However, it is not the goal of this paper to study any of these models in detail or propose or validate a specific model as the "right" estimation models for modifiability.

6.1 Issues in the Definition of External Attributes

While the distinction between internal and external software attributes may be useful to understand their nature, we would like to point out a few issues with this distinction.

No such definition in Measurement Theory. The distinction between internal and external attributes can only be found in the Software Measurement literature (e.g., [16,15], but not in the general, standard, authoritative literature on Measurement [20,34]

Incompleteness of the Definition. The definition of a measure given by Measurement Theory [20,34] is the one reported in Section 3.1: a measure is a function that associates a value with an entity. So, it is knowledge from that entity *alone* that must used in the definition of the measure, and not other entities that belong to the "environment" of the entity.

Logical Problems in Defining Attributes by Means of their Measures. The distinction between internal and external software attributes is based on whether their measures can be based on the entities alone or an "environment" as well, although attributes exist prior to and independent of how they can be measured. However, the definition of a measure logically *follows* the definition of the attribute it purports to measure: one defines a measure based on the attribute, not the attribute based on the measure. Also, suppose that two measures are defined for an attribute: one takes into account only information from the entity being measured, while the other also takes into account additional information about the "environment." According to the former measure, the attribute would be an internal one, but an external one according to the latter. So, the nature of the attribute would be uncertain, to say the least.

Deterministic vs. Probabilistic Approaches. An external software attribute (e.g., reliability or maintainability) may be affected by many variables (the "environment") in addition to the specific entity, so it would not be sensible to build a deterministic measure for it.

Using Aggregate Indicators. Aggregate indicators are often used to quantify external software attributes. For instance, the Mean Time Between Failures ($MTBF$) may be a quite useful piece of information about reliability, but it is not a measure of reliability in itself as we now explain.

- $MBTF$ is the expected value of the probability distribution of the time between failures, so quantifying $MBTF$ implies knowing this probability distribution. However, this is impossible, since probabilities cannot be measured in a frequentist approach, but they can only be estimated. This implies that $MBTF$ itself can only be estimated, but not measured.
- The probability distribution is a conditional one anyway, since it depends on the environment in which the program is used.

Validating a Probabilistic Representation for an Attribute. Probability Representations can be empirically validated in a probabilistic sense, while deterministic representations (like the ones shown in Section 3) should be validated in a totally different way. For instance, to check whether software size is additive with respect to some kind of concatenation operation, one should take all possible program segments, make all possible concatenations, and check if for all of these concatenations size is truly additive–which is totally unfeasible. Probability Representations can be validated through statistical inference procedures. It is true that these procedures can never provide absolute certainty, but this is acceptable because of the random nature of the modeling.

6.2 Probability Representations in Measurement Theory

Here, we introduce the basic concepts of Probability Representations defined in Measurement Theory [20] by slightly adapting them to our Software Measurement case. We first need to introduce the concept of algebra and of σ-algebra of sets on a set X.

Definition 10 (Algebra on a Set). *Suppose that X is a nonempty set, and that E is a nonempty family of subsets of X. E is an algebra of sets on X if and only if, for every $A, B \in E$*

1. $X - A \in E$
2. $A \bigcup B \in E$.

The elements of E are called events and the individual elements of X are called outcomes, each of which is a possible results of a so-called random experiment [19]. So, an event is actually a set of outcomes, and X is the set of all possible outcomes.

Definition 11 (The Concept of σ-Algebra on a Set). *If the conditions in Definition 10 hold and, in addition, E is closed under countable unions, i.e., whenever $A_i \in E$, with $i = 1, 2, \ldots$, it follows that $\bigcup_{i=1}^{\infty} A_i \in E$, then E is called a σ-algebra on X.*

Based on these definitions, the usual axiomatic definition of *unconditional* probability can be given [20].

However, we are here interested in *conditional* probability representations, because we are interested in conditional probabilities like the following ones.

Continuous case. $P(Eff \leq eff|art, env)$, i.e., the probability that a specified event occurs if one uses an amount of effort that is at most eff, provided that the environment *env* in which it happens and the artifact *art* on which it happens are specified, e.g., the probability that a specified artifact *art* is modified correctly with at most a specified amount of effort eff, in a specified modification environment *env*. In this case, effort Eff is the random variable, once the environment and the artifact are known (i.e., conditioned on their knowledge). This probability can be used to quantify the external software attribute "modifiability," for which we provide a model in Section 6.3.

Discrete case. $P(N \leq n|art, env)$, i.e., the probability that a specified event occurs after at most n trials, provided that the environment *env* in which it occurs and the artifact *art* on which it occurs are specified, e.g., the probability that a specified program *art* is covered (according to some specified notion of coverage) by executing it with at most n inputs, in a specified environment *env*. The number of trials N is the random variable, once the environment and the artifact are known (i.e., conditioned on their knowledge). This probability can be used to quantify the external software attribute "coverability." More details are provided in [24].

The set-theoretic notation of the theory of [20] can be interpreted as follows for our goals. The set X is the set of all possible triple of the form $< op, art, env >$ where

- *op* is an "observable phenomenon," i.e., built via a predicate like $Eff \leq eff$ or $N \leq n$
- *art* is a specific artifact (e.g., a program)
- *env* is an environment in which *art* is used and *op* is observed.

For notational convenience, we denote conditional probabilities as $P(op|art, env)$. For instance, reliability can be quantified as a conditional probability, as follows

$$R(t) = P(t \leq T|art, env)$$

i.e., the probability that a failure occurs at time T not less than a specified time t, in a specified program *art* and in a specified operational environment *env*.

Like with deterministic representations, we capture our intuitive knowledge on the ordering among conditional events via an order relation $\succsim$, whose meaning is "qualitatively at least as probable as" [20]. In general, suppose that A, B, C, and D are events. By writing $A|B \succsim C|D$, we mean that event A, when event B is known to occur, is "qualitatively at least as probable as" event C, when event D is known to occur. In other words, instead of having a deterministic ordering among entities according to some attribute of interest like in deterministic representations, one has a probabilistic ordering. For instance, one may order software programs according to their modifiability in a probabilistic way (i.e., a program in an operational environment is *qualitatively at least as modifiable as* another program in another operational environment), instead of a deterministic way (i.e., a program is *certainly* more modifiable than another). For completeness, based on relation $\succsim$, one may also define relation $\sim$ as follows: $A|B \sim C|D$ if and only if $A|B \succsim C|D$ and $C|D \succsim A|B$.

The Representation Condition needed for conditional probability representations is

$$A|B \succsim C|D \Leftrightarrow P(A|B) \geq P(C|D)$$

At a first glance, it may appear that the order relation $\succsim$ is a binary relation that is a subset of $(E \times E) \times (E \times E)$, since $A|B \succsim C|D$ is simply a graphical convention for $< A, B > \succsim < C, D >$. However, some caution must be exercised. Conditional probabilities are defined as $P(A|B) = P(A \cap B)/P(B)$, so $P(A|B)$ is defined only if $P(B) > 0$. Thus, if $P(B) = 0$, writing $A|B \succsim C|D \Leftrightarrow P(A|B) \geq P(C|D)$ makes no sense. This means that any event B such that $P(B) = 0$ cannot appear as the second element of $A|B$, i.e., as the conditioning event. Thus, by denoting with NN (as in $NonNull$) the set of events B such that $P(B) > 0$, the order relation $\succsim$ is actually a binary relation on $E \times NN$.

Here are necessary conditions (slightly adapted from [20]) for the Representation Condition. We simply list these axioms here for completeness. At any rate, more details on the general theoretical approach are provided in [20], and on their application in Software Measurement in [24].

Definition 12 (Conditional Probability Axioms). *Let X be a nonempty set, E an algebra of sets on X, NN a subset of E, and $\succsim$ a binary relation on $E \times NN$. The quadruple $< X, E, NN, \succsim >$ is a structure of qualitative conditional probability if and only if for every A, B, C, A', B', and $C' \in E$ (or $\in NN$, whenever the name of the event appears to the right of $'|'$), the following axioms hold.*

1. *$< E \times NN, \succsim >$ is a weak order.*
2. *$X \in NN$ and $A \in E - NN$ if and only if $A|X \sim \emptyset|X$.*
3. *$X|X \sim A|A$ and $X|X \succsim A|B$.*
4. *$A|B \sim A \bigcap B|B$.*

5. *Suppose that* $A \bigcap B = A' \bigcap B' = \emptyset$. *If* $A|C \gtrsim A'|C$ *and* $B|C \gtrsim B'|C'$*, then* $A \bigcup B|C \gtrsim A' \bigcup B'|C'$*; also, if either hypothesis is* $\succ$*, then the conclusion is* $\succ$*.*
6. *Suppose that* $A \supset B \supset C$ *and* $A' \supset B' \supset C'$*. If* $B|A \gtrsim C'|B'$ *and* $C|B \gtrsim B'|A'$*, then* $C|A \gtrsim C'|A'$*; moreover, if either hypothesis is* $\succ$*, then the conclusion is* $\succ$*.*

The above axioms can be used as necessary conditions to find additional conditions under which an ordering relation on E has an order-preserving function P that satisfies the above axioms, i.e., the Representation Condition of Section 3.1. This means that different probability representations, i.e., different probability functions, may exist so that the Representation Condition of Section 3.1 is satisfied. Additional conditions may be provided to make the set of axioms sufficient [20]. However, we are not interested in these additional conditions here. In Section 6.3, we show how to build actual probability functions.

Note that it is not important here that our probabilistic intuitive knowledge is accurate. We only need to put the concept of using probabilities for external software attributes on solid bases. Empirical studies will show whether our intuitive knowledge is correct. If it is not, we need to modify our intuitive knowledge in such a way as to fit the empirical results. This is another value added of this approach, since it allows us to increase and refine our empirical knowledge about an attribute of interest.

6.3 Representing Modifiability

Based on the above Probability Representation approach, modifiability is here quantified as the probability that a given artifact, in a specified modification environment, is modified with a specified amount of effort, i.e., $Mod(eff) = Mod(Eff \leq eff|art, env) = P(Eff \leq eff|art, env)$.

Simply to show how a modifiability model can be built [24], rather than proposing it as the "right" or "preferred" modifiability model, suppose that the modifiability rate of an artifact (which is the counterpart of the hazard rate used for reliability [27] when studying modifiability) is a linear function of the probability $Mod(eff)$ that the artifact has been modified with eff effort. The underlying idea is that, if an artifact needs to undergo one specific modification, As more and more effort is used to carry out that modification, (1) the higher is the probability $Mod(eff)$ that the modification is actually carried out, and (2) the higher is the instantaneous probability that the modification is going to be carried out if it has not been carried out so far (this instantaneous probability is actually the modifiability rate). Thus, we can write

$$\frac{Mod'(eff)}{1 - Mod(eff)} = a + bMod(eff)$$

Coefficient a is the initial modifiability rate at $eff = 0$ and coefficient b describes how well one uses the information contained in the modification activities up to effort eff. It can be shown that the function that describes the modification probability in closed form is

$$Mod(eff) = \frac{a(e^{(a+b)eff} - 1)}{b + ae^{(a+b)eff}}$$

Parameters a and b may be explicitly related to *env* and *art*. For instance, they may be a function of the number of people that modify the artifact and the size (e.g., the number of lines of code) of the artifact. For instance, we could have $a = \alpha \cdot \#people$ and $b = \beta \cdot 1/LOC$. Based on the past history of efforts needed to modify an artifact, parameters a and b or α and β are estimated by using some statistical techniques [19]–provided that $a + b > 0$, and $a > 0$, since the modifiability rate is positive.

Once the probability distribution is known, a number of derived indices may be used to provide a concise idea for the probability distribution of an attribute, e.g., the expected values for the distributions obtained for modifiability (i.e., the average effort needed for modifying a software artifact). These derived indices may be used for instance to set process goals (e.g., the average effort needed for modifying a software artifact must be no greater than a specified value) or compare competing techniques (e.g., given two development techniques, one may choose the one that has the lower average average effort needed to modify a program).

One final note on modifiability. One may very well argue that modifiability depends on the specific modification that needs to be carried out. However, a similar remark applies to software reliability, which clearly depends on the specific inputs selected or the selection policy used. The fact that there are several different modifications that may be carried out and one of them is actually carried out in a specific way according to a random policy mirrors the random selection of inputs that is used in software reliability modeling. Actually, in software reliability modeling, one may argue that, once an input has been selected for a deterministic program, then there is only one possible result which is either correct or incorrect. Instead, when it comes to modifiability, when the need for a modification has been identified, many different random variables may influence the way the actual modification is carried out, and therefore the effort needed. Thus, the use of a probabilistic model may be even more justified for modifiability than for reliability.

7 GQM/MEDEA

As the definition of a measure needs to be carried out carefully, it is necessary to have a defined process in place. In this section, we describe the GQM/MEasure DEfinition Approach (GQM/MEDEA) [12], which takes advantage of the goal-oriented nature of the Goal/Question/Metric paradigm [6] to set the measurement goals of any measurement activity to guide the measure definition and validation process. GQM/MEDEA can be used for building so-called predictive models, i.e., models that use one or more internal software attributes to predict an external software attribute or process attribute of interest. We use a semi-formal notation, Data Flow Diagrams (DFDs) [14], to define and refine the steps used in GQM/MEDEA. In DFDs, bubbles denote activities, boxes external information sources/sinks, and arrows data flows. The arrows also provide an idea of the order in which the activities are executed, though, during the execution of an activity, any other activity may be resumed or started as long as its inputs are available. A bubble may be refined by a DFD, provided that the incoming and outgoing data flows of the bubble and its refining DFD are the same.

The topmost diagram in DFDs is called the Context Diagram (shown in Fig. 3, which represents the entire process as one bubble and shows the interactions of the measure definition process with information sources and sinks.

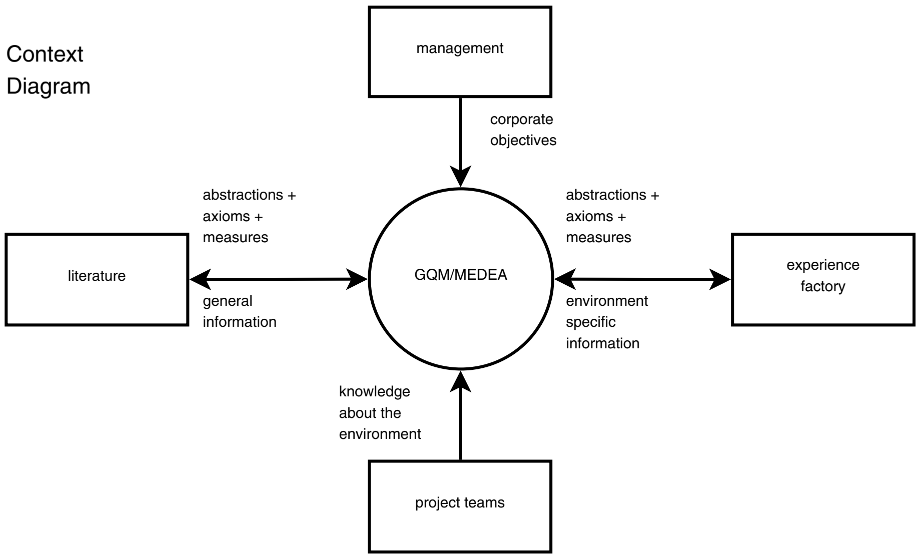
Context Diagram
management
corporate objectives
abstractions + axioms + measures
literature
general information
GQM/MEDEA
abstractions + axioms + measures
experience factory
environment specific information
knowledge about the environment
project teams

Fig. 3. Interactions of GQM/MEDEA with information sources and sinks

Several sources of information, as shown in Fig. 3, are used by the GQM/MEDEA process:

– the management, to help define measures that are useful to achieve the corporate goals of a software organization (e.g., "reduce maintenance effort");
– the personnel of the project(s) used to practically validate the measures; the people involved in the empirical study provide important information about the context of the study that cannot be found anywhere else;
– experience belonging to the software organization that has been previously gathered, distilled, and stored in the experience factory [4,7,5] (e.g., quantitative prediction models, lessons learned from past projects, measurement tools and procedures, or even raw project data);
– the scientific literature.

The measurement process itself should contribute its outputs to the experience factory with new artifacts, in the form of abstractions (i.e., models of software artifacts), measure properties, and measures. These outputs should be packaged and stored so that they can be efficiently and effectively reused later on, thus reducing the cost of measurement in an organization [6]. In a mature development environment, inputs for most of the steps should come from reused knowledge. Some of the steps that are made explicit in GQM/MEDEA are often left implicit during the definition of a measure. We have made them explicit to show all the logical steps that are carried out to identify all potential sources of problems. The main contribution of GQM/MEDEA to GQM is the definition of an organized process for the definition of software product measures based on GQM goals.

Fig. 4 7 shows the high-level structure of the approach.

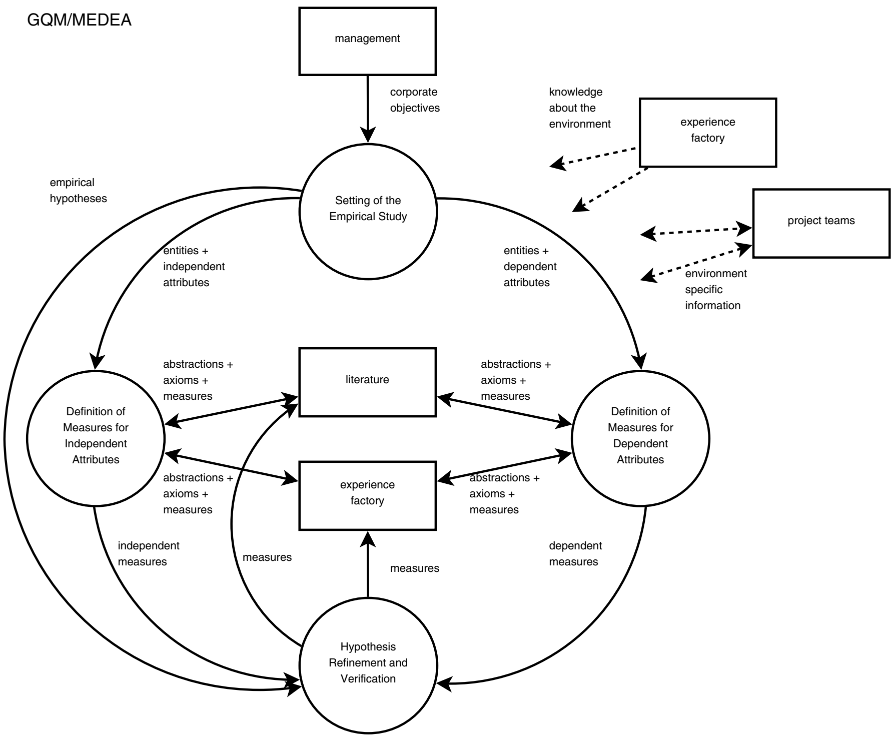
GQM/MEDEA
management
corporate objectives
knowledge about the environment
experience factory
project teams
environment specific information
empirical hypotheses
Setting of the Empirical Study
entities + independent attributes
entities + dependent attributes
abstractions + axioms + measures
literature
abstractions + axioms + measures
Definition of Measures for Independent Attributes
Definition of Measures for Dependent Attributes
abstractions + axioms + measures
experience factory
abstractions + axioms + measures
independent measures
measures
measures
dependent measures
Hypothesis Refinement and Verification

Fig. 4. GQM/MEDEA: high-level structure

Each high-level step of Fig. 4 is refined in the more detailed DFDs of Fig. 5. In Fig. 4 and in Fig. 5, we do not show explicitly the environment-specific information from the project teams and the experience factory, which permeates all activities represented in these figures, not to clutter the diagrams.

We now concisely illustrate the steps in Fig. 5. The interested reader may refer to [12] for more detailed information about GQM/MEDEA.

7.1 Setting of the Empirical Study

The steps and their connections are in Fig. 5(a). Corporate objectives (e.g., "reduce maintenance effort") are first refined into tactical goals (e.g., "improve the maintainability of the final product"), and then tactical goals are refined into measurement goals (e.g., "predict the maintainability of software code based on its design"). These refinements are based on knowledge about the environment provided by the project teams and the experience factory, which help identify processes and products that measurement should address. As the measurement goal should be made as precise as possible, goal-oriented techniques [6] can be used to detail the object of study, the specific quality to be investigated, the specific purpose for which the quality should be investigated, the immediate beneficiaries of the empirical investigation (e.g., the project managers), and the specific context in which the empirical investigation is carried out.

The measurement goals help establish a set of empirical hypotheses that relate (independent) attributes of some entities (e.g., the coupling of software components in design) to other (dependent) attributes of the same or different entities (e.g., the maintainability of the maintained software code). Dependent attributes are usually 1) external quality attributes of software systems or parts thereof, e.g., reliability, maintainability, effort, or 2) process attributes, e.g., development effort, development time, or number of faults. Independent attributes capture factors that are usually hypothesized to have a causal relationship with the dependent attribute. An empirical hypothesis describes how these two attributes are believed to be related, e.g., the coupling of the modules identified via a product's design is hypothesized to be negatively related to the final code maintainability. Empirical hypotheses cannot describe a specific functional form for this hypothesized dependency, because no measures have been yet defined for the independent and the dependent attributes. These definitions are carried out in the remainder of the measure definition process. So, empirical hypotheses are not statistical ones and cannot be tested. However, in the last phase of the GQM/MEDEA process (see Section 7.4), when specific measures have been defined for the independent and the dependent attributes, empirical hypotheses will be instantiated into statistical (and therefore testable) hypotheses.

7.2 Definition of Measures for the Independent Attributes

The process used to define measures for independent attributes is in Fig. 5(b). Independent attributes are formalized to characterize their measures, in ways like those in Sections 3 and 4. If an axiomatic approach is chosen, it is necessary to formalize entities via abstractions (e.g., graph models), which are built based on the entities, the independent attributes and their defining axioms. Once a correct abstraction is built, the axioms can be instantiated, i.e., a precise mapping of the specific characteristics of the model can be done onto the characteristics of

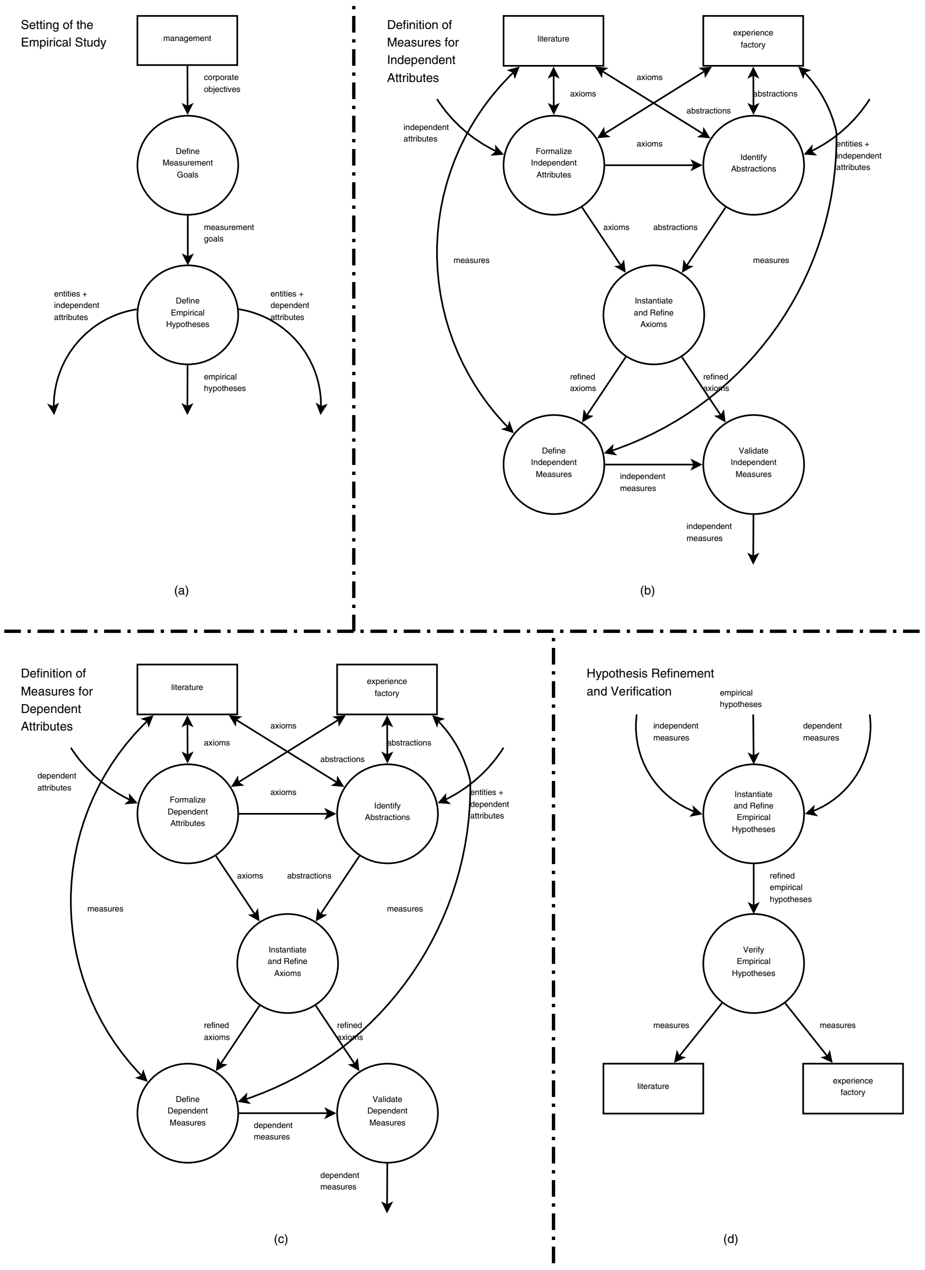
Setting of the Empirical Study
management
corporate objectives
Define Measurement Goals
measurement goals
Define Empirical Hypotheses
entities + independent attributes
entities + dependent attributes
empirical hypotheses
(a)
Definition of Measures for Independent Attributes
literature
experience factory
axioms
abstractions
independent attributes
Formalize Independent Attributes
Identify Abstractions
entities + independent attributes
measures
Instantiate and Refine Axioms
refined axioms
Define Independent Measures
Validate Independent Measures
independent measures
(b)
Definition of Measures for Dependent Attributes
dependent attributes
Formalize Dependent Attributes
entities + dependent attributes
Define Dependent Measures
Validate Dependent Measures
dependent measures
(c)
Hypothesis Refinement and Verification
empirical hypotheses
independent measures
dependent measures
Instantiate and Refine Empirical Hypotheses
refined empirical hypotheses
Verify Empirical Hypotheses
measures
literature
experience factory
(d)

Fig. 5. GQM/MEDEA: refined structure

the mathematical model upon which the formalization of an attribute is based. For instance, an abstraction of an object-oriented system can be obtained by mapping each class onto a different element of a graph-based model and each dependence between classes onto a relationship, like in Section 4. If a Measurement Theory-based approach is used, we need to identify the entities, the relationships we intuitively expect among entities, and the composition operations between entities first. In other words, we need to build the Empirical Relational System first, and the Numerical Relational System later on. Note that using an axiomatic approach may not provide all the information that is needed to build a measure. Different measures, which will give different orderings of entities, can be defined that satisfy a set of axioms. For instance, LOC and $\#Statements$ both satisfy the axioms for size. However, given two program segments, it may very well be that a program segment has a value for LOC greater than the other program segment, but a smaller value for $\#Statements$. So, an axiom set may be incomplete, and additional properties may need to be introduced to refine it and obtain a complete ordering of entities. These additional properties will depend on the specific application environment. Based on this refined set of axioms, new measures are defined or existing ones are selected for the attributes of entities. Additional checks may be required to verify whether the defined measures really comply with the refined set of axioms.

7.3 Definition of Measures for the Dependent Attributes

The GQM/MEDEA approach deals with independent and dependent attributes of entities in much the same way, as can be seen from Fig. 5(c). For instance, if our dependent attribute is a process attribute like maintenance effort, then effort can be modeled as a type of size. If our dependent attribute is maintainability, then we can use a Probability Representation approach like the one illustrated in Section 6.2. In the context of experimental design, the definition of measures for independent and dependent attributes via an organized and structured approach has the goal of reducing the threats to what is referred to as construct validity [37], i.e., the fact that a measure adequately captures the attribute it purports to measure. Although construct validity is key to the validity of an experiment, few guidelines exist to address that issue.

7.4 Hypothesis Refinement and Verification

Fig. 5(d) shows the steps carried out for hypothesis refinement and verification. The empirical hypotheses established as shown in Section 7.1 need to be refined and instantiated into statistical hypotheses to be verified, by using the measures defined for the independent and dependent attributes. One possibility is to provide a specific functional form for the relationship between independent and dependent measures, e.g., a linear relationship, so a correlation would be tested in a statistical way, based on actual development data. (Statistically testing associations would not be sufficient, because this does not lead to a prediction model, as we assume in this section for validating software measures.) Typically, additional data analysis problems have to be addressed such as outlier analysis [3] or the statistical power [19] of the study. The predictive model can be used to verify the plausibility of empirical hypotheses, in addition to being used as a prediction model in its own right.

8 Conclusions and Future Work

In this paper, we have shown a number of approaches for dealing with the fundamental aspects of Software Measurement, by describing the notions of Measurement Theory for both internal and external software attributes, the definition of properties for software measures via Axiomatic Approaches, and the proposal of an integrated process where the foundational aspects of Software Measurement can be coherently used.

A number of research and application direction should be pursued, including

- using Measurement Theory for modeling internal and external software attributes in a way that is consistent with intuition
- refining Axiomatic Approaches by building generalized consensus around the properties for software attributes
- extending Axiomatic Approaches to other software attributes of interest and understanding the relationships between different software attributes
- defining and refining processes for using the foundational aspects of Software Measurement in practice.

Acknowledgments. This work has been partially supported by project "Metodi e tecniche per la modellazione, lo sviluppo e la valutazione di sistemi software" funded by Università degli Studi dell'Insubria.

References

1. ISO/IEC 9126-1:2001- Software Engineering - Product Quality Part 1: Quality Model. ISO/IEC (2001)
2. ISO/IEC 9126-2:2002- Software Engineering - Product Quality Part 1: External Metrics. ISO/IEC (2002)
3. Barnett, V., Lewis, T.: Outliers in statistical data, 3rd edn. John Wiley & Sons (1994)
4. Basili, V.R.: The Experience Factory and Its Relationship to Other Improvement Paradigms. In: Sommerville, I., Paul, M. (eds.) ESEC 1993. LNCS, vol. 717, pp. 68–83. Springer, Heidelberg (1993)
5. Basili, V.R., Caldiera, G., Rombach, H.D.: The Experience Factory. Encyclopedia of Software Engineering, vol. 2, pp. 511–519. John Wiley & Sons (2002), `http://books.google.es/books?id=CXpUAAAAMAAJ`
6. Basili, V.R., Rombach, H.D.: The tame project: Towards improvement-oriented software environments. IEEE Transactions on Software Engineering 14(6), 758–773 (1988)

7. Basili, V.R., Zelkowitz, M.V., McGarry, F.E., Page, G.T., Waligora, S., Pajerski, R.: Sel's software process improvement program. IEEE Software 12(6), 83–87 (1995)
8. Briand, L.C., Daly, J.W., Wüst, J.: A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering 3(1), 65–117 (1998)
9. Briand, L.C., Differding, C., Rombach, H.D.: Practical guidelines for measurement-based process improvement. Software Process: Improvement and Practice 2(4), 253–280 (1996), `http://www3.interscience.wiley.com/journal/24853/abstract`
10. Briand, L.C., Morasca, S., Basili, V.R.: Property-based software engineering measurement. IEEE Transactions on Software Engineering 22, 68–86 (1996), `http://portal.acm.org/citation.cfm?id=229713.229722`
11. Briand, L.C., Morasca, S., Basili, V.R.: Defining and validating measures for object-based high-level design. IEEE Transactions on Software Engineering 25, 722–743 (1999), `http://portal.acm.org/citation.cfm?id=325392.325404`
12. Briand, L.C., Morasca, S., Basili, V.R.: An operational process for goal-driven definition of measures. IEEE Transactions on Software Engineering 28, 1106–1125 (2002), `http://portal.acm.org/citation.cfm?id=630832.631301`
13. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
14. DeMarco, T.: Structured analysis and system specification. Yourdon computing series. Yourdon, Upper Saddle River (1979)
15. Fenton, N., Pfleeger, S.L.: Software metrics: a rigorous and practical approach, 2nd edn. PWS Publishing Co., Boston (1997)
16. Fenton, N.E.: Software metrics - a rigorous approach. Chapman and Hall (1991)
17. Halstead, M.H.: Elements of software science. Operating and programming systems series. Elsevier (1977), `http://books.google.com/books?id=zPcmAAAAMAAJ`
18. Henry, S.M., Kafura, D.G.: Software structure metrics based on information flow. IEEE Transactions on Software Engineering 7(5), 510–518 (1981)
19. Kendall, M.G., Stuart, A.: The advanced theory of statistics, 4th edn. C. Griffin, London (1977)
20. Krantz, D.H., Luce, R.D., Suppes, P., Tversky, A.: Foundations of Measurement, vol. 1. Academic Press, San Diego (1971)
21. Lakshmanan, K.B., Jayaprakash, S., Sinha, P.K.: Properties of control-flow complexity measures. IEEE Transactions on Software Engineering 17(12), 1289–1295 (1991)
22. McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering 2(4), 308–320 (1976)
23. Morasca, S.: Refining the axiomatic definition of internal software attributes. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, Kaiserslautern, Germany, October 9-10, pp. 188–197. ACM, New York (2008), `http://doi.acm.org/10.1145/1414004.1414035`
24. Morasca, S.: A probability-based approach for measuring external attributes of software artifacts. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, Lake Buena Vista, FL, USA, October 15-16, pp. 44–55. IEEE Computer Society, Washington, DC (2009), `http://dx.doi.org/10.1109/ESEM.2009.5316048`

25. Morasca, S., Briand, L.C.: Towards a theoretical framework for measuring software attributes. In: Proceedings of the 4th International Symposium on Software Metrics, IEEE METRICS 1997, Albuquerque, NM, USA, November 5-7, pp. 119–126. IEEE Computer Society, Washington, DC (1997), http://portal.acm.org/citation.cfm?id=823454.823906
26. Musa, J.D.: A theory of software reliability and its application. IEEE Transactions on Software Engineering 1(3), 312–327 (1975)
27. Musa, J.D.: Software Reliability Engineering. Osborne/McGraw-Hill (1998)
28. Oman, P., Hagemeister, J.R.: Metrics for assessing a software system's maintainability. In: Proceedings of ICSM 1992, Orlando, FL, USA, pp. 337–344 (1992)
29. Oviedo, E.I.: Control flow, data flow and program complexity. In: Proceedings of the 4th Computer Software and Applications Conference, COMPSAC 1980, Chicago, IL, USA, October 27-31, pp. 146–152. IEEE Press, Piscataway (1980)
30. Pap, E.: Some elements of the classical measure theory, pp. 27–82. Elsevier (2002), http://books.google.je/books?id=LylS9gsFEUEC
31. Poels, G., Dedene, G.: Comments on property-based software engineering measurement: Refining the additivity properties. IEEE Transactions on Software Engineering 23(3), 190–195 (1997)
32. Poels, G., Dedene, G.: Distance-based software measurement: necessary and sufficient properties for software measures. Information & Software Technology 42(1), 35–46 (2000)
33. Prather, R.E.: An axiomatic theory of software complexity measure. The Computer Journal 27(4), 340–347 (1984)
34. Roberts, F.: Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences, Encyclopedia of Mathematics and its Applications, vol. 7. Addison-Wesley (1979)
35. Sauro, J., Kindlund, E.: A method to standardize usability metrics into a single score. In: CHI 2005: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Portland, Oregon, USA, pp. 401–409 (2005)
36. Shepperd, M.J.: Foundations of software measurement. Prentice Hall (1995)
37. Spector, P.E.: Research designs. Quantitative applications in the social sciences. Sage Publications (1981), http://books.google.com/books?id=NQAJE_sh1qIC
38. Stevens, S.S.: On the theory of scales of measurement. Science 103(2684), 677–680 (1946), http://www.ncbi.nlm.nih.gov/pubmed/16085193
39. Weyuker, E.J.: Evaluating software complexity measures. IEEE Transactions on Software Engineering 14(9), 1357–1365 (1988)
40. Yourdon, E., Constantine, L.L.: Structured design: fundamentals of a discipline of computer program and systems design, 2nd edn. Yourdon Press, New York (1978)