An Operational Process for Goal-Driven Definition of Measures

Lionel C. Briand
Carleton University
Systems and Computer Engineering.
1125 Colonel By Dr.
Canada, ON, K1S 5B6
briand@sce.carleton.ca

Sandro Morasca
Dip. di Scienze
Chimiche, Fisiche e Matematiche
Università degli Studi dell'Insubria
Via Valleggio 11
I-22100 Como, Italy
sandro.morasca@uninsubria.it

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

Abstract

We propose an approach (GQM/MEDEA) for defining measures of product attributes in software engineering. The approach is driven by the experimental goals of measurement, expressed via the GQM paradigm, and a set of empirical hypotheses. To make the empirical hypotheses quantitatively verifiable, GQM/MEDEA supports the definition of theoretically valid measures for the attributes of interest based on their expected mathematical properties. The empirical hypotheses are subject to experimental verification. This approach integrates several research contributions from the literature into a consistent, practical, and rigorous approach.

Keywords: Software measurement, Software quality, Goal-Question-Metric Paradigm

1. Introduction

Measures can help address some of the most critical issues in software development and provide support for planning, monitoring, controlling, and evaluating the software process. In the recent literature, a large number of measures have appeared for capturing software product attributes in a quantitative way. However, few measures have successfully survived the initial definition phase and are actually used in the industry. This is due to a number of problems related to the theoretical and empirical validity of many measures, the most relevant of which are summarized next.

- Measures are not always defined in the context of some explicit and well-defined measurement goal derived from an objective of industrial interest they help reach, e.g., reduction of development effort or faults present in the software products.
- Even if the goal is made explicit, the experimental hypotheses are often not made explicit, e.g., what do you expect to learn from the analysis?
- Measurement definitions do not always take into account the environment or context in which they will be applied, e.g., would you use a complexity measure that was defined for non-object oriented software in an object-oriented context?
- A reasonable theoretical validation of the measure is often not possible because the attribute that a measure aims to quantify is often not well defined, e.g., are you using a measure of complexity (attribute) that clearly models your intuition about complexity?
- A large number of measures have never been subject to an empirical validation, e.g., how do you know which measures of size predict effort best in your environment?

This situation has frequently led to some degree of fuzziness in the measure definitions, properties, and underlying assumptions, making the use of the measures difficult, their interpretation hazardous, and the results of the various validation studies somewhat contradictory [29, 31].

The above problems are inherent to any young discipline, especially one that is human intensive. Software measurement is currently in a phase in which terminology, principles, and methods are still being defined and consolidated. The human-intensive nature of software engineering makes its measurement somewhat closer to that of the social sciences rather than the physical sciences. The phenomena that are studied involve a number of variables that depend on human behavior and cannot be controlled easily. We should not expect to find quantitative laws that are generally valid and applicable, and have the same precision and accuracy as the laws of Physics, for instance. As a consequence, the identification of universally valid and applicable measures may be an ideal, long term research goal, which cannot be achieved in the near future, if at all.

These characteristics do not imply that quantitative progress cannot be made in the software measurement field. On the contrary, a disciplined approach to the definition of a measure allows practitioners and researchers to (1) build upon a solid theoretical basis, (2) link the measure to the application at hand, (3) provide a clearer rationale of the underlying definition of a measure and its applications, (4) judge whether it is necessary to define a new measure or instead reuse an existing one for a specific application, and (5) interpret the results of an experiment or a case study, especially when one does not obtain the expected results.

This paper introduces (based on our experience [12, 17, 9, 16, 20, 19]) a measure definition process (GQM/MEDEA: GQM/MEtric DEfinition Approach), usable as a practical *guideline* to design and reuse technically sound and useful measures. The focus here is the construction of *prediction systems*, i.e., models that establish a correspondence between measures for software attributes. One of these measures quantifies the dependent variable and is usually related to a product or process attribute of industrial interest (e.g., software fault-proneness, cost, time-to-market). The other measures quantify various product and process attributes (e.g., coupling of the components) and are the independent variables of the model. Based on knowledge of the application environment, an explicit definition of the specific goal(s) to be addressed, and a set of experimental hypotheses that need to be validated, we identify attributes of interest and define theoretically valid measures for them. These measures are subsequently used to validate the experimental hypotheses.

Prediction systems are a crucial application of measurement as evidenced by the industry driven, ISO/IEC standard [30] on software product quality. The standard states that internal product measures should be related to external product quality in order to be useful and meaningful. This implies that a prediction model must be built to explain the relationship between internal product measures and external quality measures.

It is our position that software engineering measurement is not about defining new measures, but about building new theories that can help solve practical problems. The value added by the definition of a new measure is not the measure itself, but the fact that there is a theory in which the new measure is used to help explain some phenomenon of interest. Our proposal shows how this theory building can be carried out, what steps are required or useful, and what intermediate (e.g., measurement goals, initial empirical hypotheses) and final results (e.g., validated hypotheses, measures) are produced.

The GQM/MEDEA approach derives from experience gathered on a number of projects in different environments. In this paper, we illustrate our measure definition approach via two industrial applications. In the first one, a well-consolidated and measurement-oriented environment was already in place, while, in the second one, measurement was a new activity for the development environment. The two case studies focus on the high-level design and specification phase, respectively, to illustrate the applicability of the approach to other phases than coding.

We advocate the need for a formal definition of the mathematical properties of measures [18]. However, the definition of a measure is itself a very human-intensive activity, which cannot be described and analyzed in a fully formal way. We expect GQM/MEDEA to be refined and tailored to fit the needs of different application contexts, as we gain experience in applying it.

Our framework takes advantage of several research contributions of the literature. Basili et al. [5, 6] have provided templates for defining operational experimental goals for software measurement. Our proposal can be seen as an extension of the GQM paradigm [2, 5, 6], which provides a mechanism for generating models, the most challenging part of the paradigm. Melton et al. have studied product abstraction properties [33]. Weyuker [42] and Tian and Zelkowitz [40] have studied desirable properties for complexity measures. Fenton and Melton [25], and Zuse [43] have investigated the use of measurement theory to determine measurement scales. Schneidewind has proposed a validation framework for measures [38].

Also, our approach draws many ideas from the theory of designing experiments and empirical studies in general [39]. GQM/MEDEA should be considered a proposal for discussion about a measure definition approach that can be accepted and used in software engineering. We believe that an intellectual process is necessary to define sound and useful software measures, supported by a solid theory which facilitates their review and refinement.

The paper is organized as follows. In the next section, we provide an overview of the GQM/MEDEA approach and a conceptual model of all the principles involved. Section 3 contains a concise description of the two application cases that we will use to illustrate our measure definition approach. Sections 4 – 7 illustrate the steps of the GQM/MEDEA approach in detail. Through a few examples, Section 8 shows how GQM/MEDEA helps identify the causes of problems that may be encountered during measure definition. Section 9 concludes and outlines the directions for future work.

2. Overview of GQM/MEDEA

We first model the steps of the proposed method using Data Flow Diagrams (DFDs) [22] and then provide a static model of the concepts we define using a UML class diagram. The former aims at showing the operational structure of GQM/MEDEA and the latter helps formalize the concepts defined and their relationships.

In DFDs, bubbles denote activities, boxes external information sources/sinks, and arrows data flows. An activity may be executed as soon as its inputs (or a subset of them, depending on the activity) are available. Thus, 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. Figure 1 shows the Context Diagram, i.e., the interactions of the measure definition process with information sources and sinks.

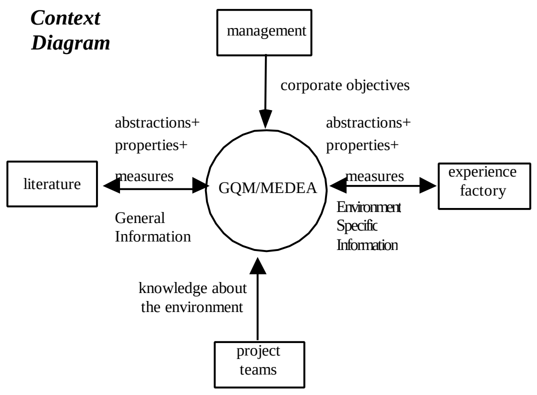

Context Diagram
management
corporate objectives
abstractions+ properties+ measures
General Information
literature
GQM/MEDEA
abstractions+ properties+ measures
Environment Specific Information
experience factory
knowledge about the environment
project teams

Figure 1. Interactions of GQM/MEDEA with information sources and sinks.

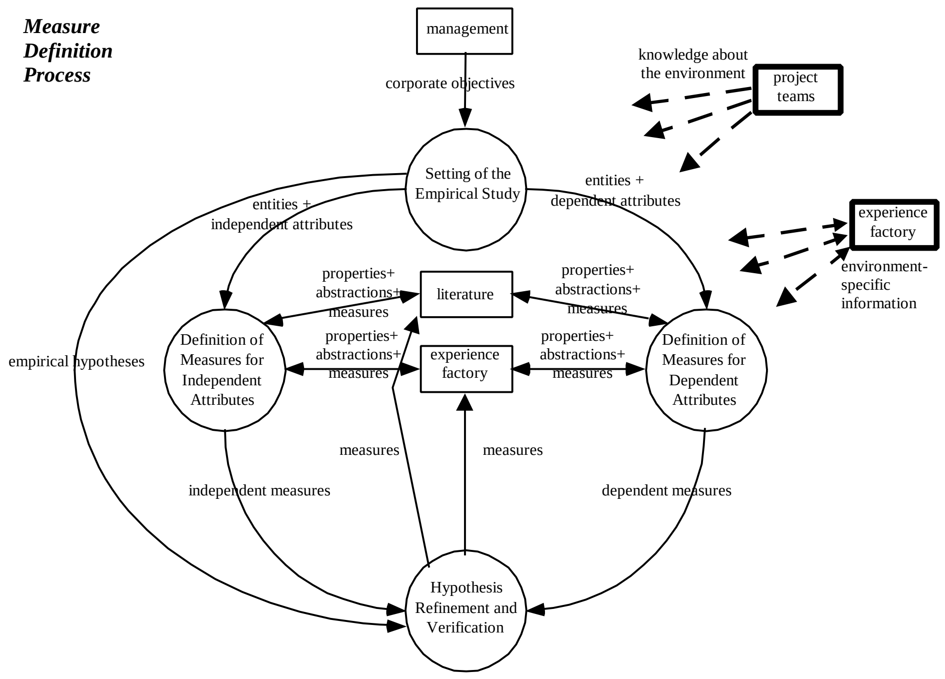
In Figure 1, the GQM/MEDEA process receives inputs from the *management*, so that the measures are defined to help the software organization achieve one or more of its corporate objectives (e.g., “reduce development costs”), at least partially.

The *project personnel* provide important information about the context of the study, as a relevant part of the knowledge is in the developers’ minds. This knowledge can be elicited in many ways (e.g., structured interviews, questionnaires) which can be rigorous and repeatable.

The experience gathered and distilled on previous projects through the *experience factory* [3, 4, 7] is an invaluable asset of the software organization. Ideally, it provides the organization with a variety of information relevant to the way an organization develops software, e.g., quantitative prediction models, lessons learned from past projects, measurement tools and procedures, or even raw project data. The measurement process itself should contribute to the experience factory with new artifacts, in the form of abstractions, measure properties, and measures, stored for later use.

The *literature* provides information about measurement programs carried out at other development sites. Properties of the attributes that need to be studied, abstractions and measures may be reused, if relevant to the current measurement program.

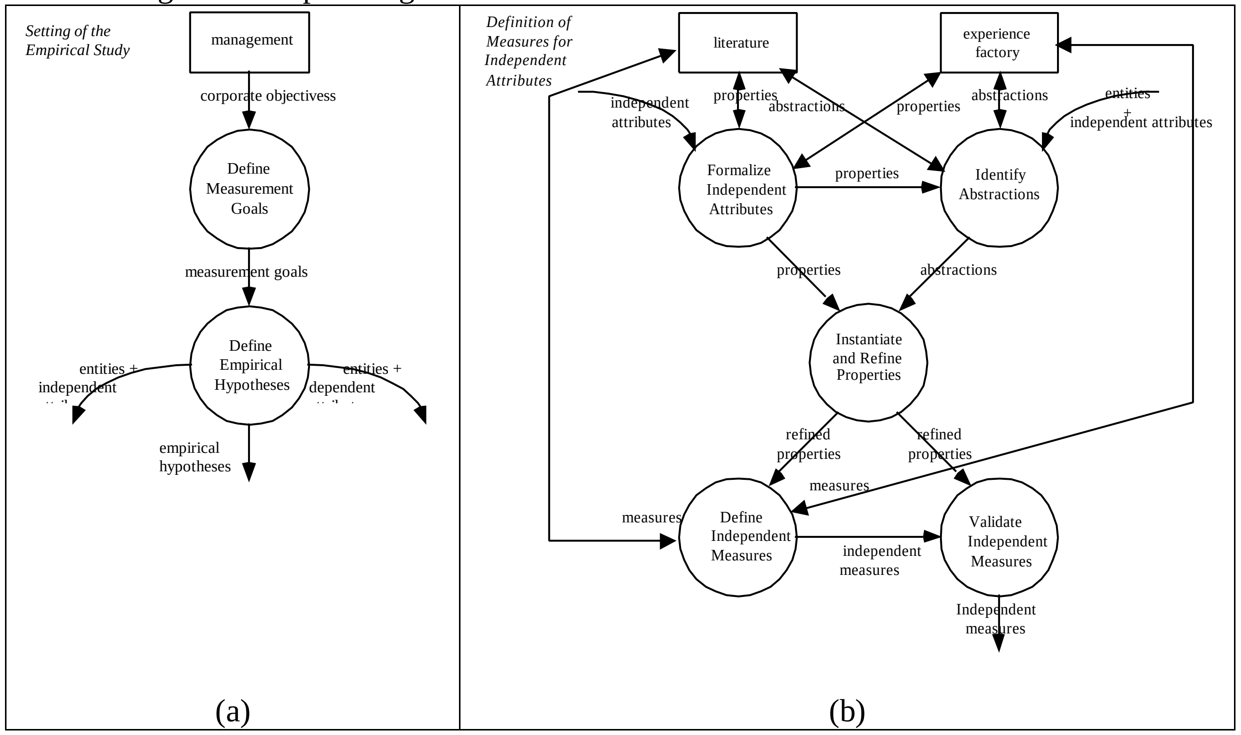
Figure 2 shows the high-level structure of the approach. Each high-level step of Figure 2 is refined in the DFDs of Figure 3. In addition to the information flows explicitly shown, environment-specific information from the project teams and the experience factory is available to almost every activity of Figures 2 and 3. We have not represented the corresponding data flows explicitly in Figures 2 and 3 since they would have cluttered the diagrams.



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

Figure 2. High-level steps of GQM/MEDEA.

The four high-level steps of Figure 2 can be described as follows.



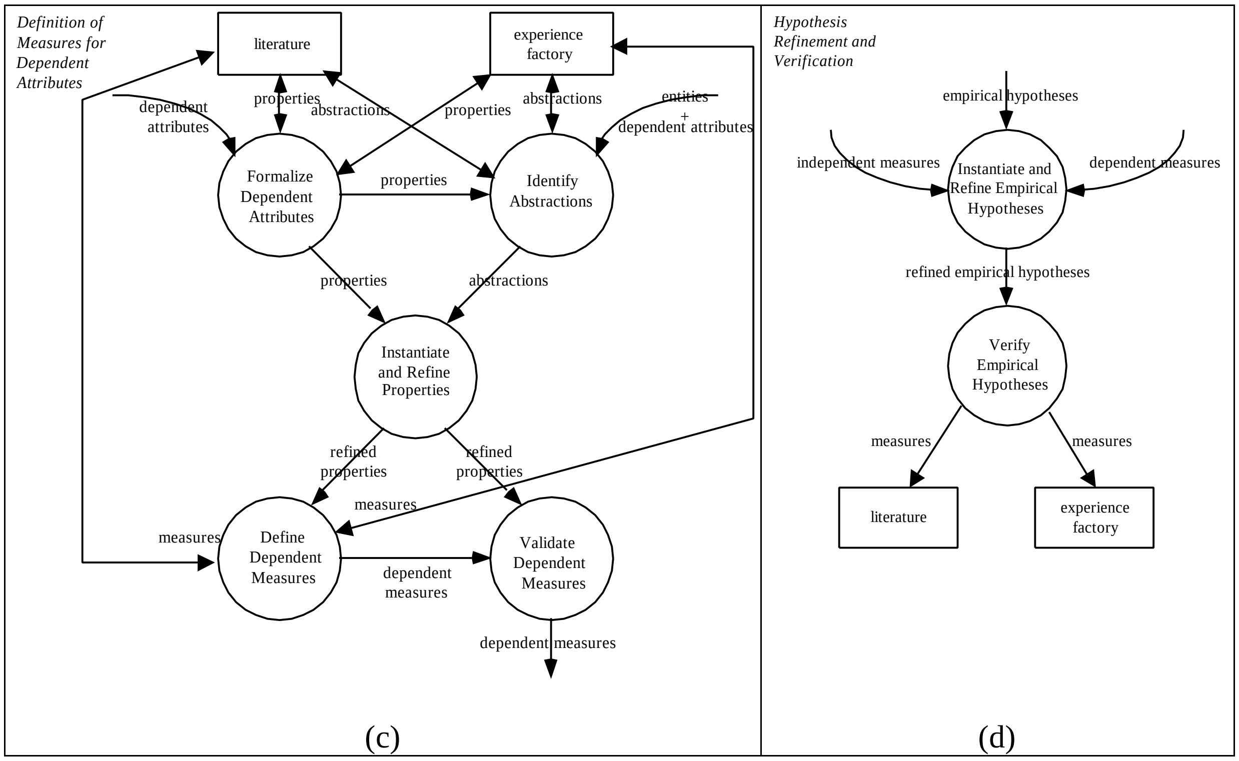


Figure 3. Refinement of the high-level steps.

1. *Setting of the empirical study* (refined in Figure 3(a) and illustrated in Section 4). Corporate objectives are refined into measurement goals, 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. Based on the measurement goals, a set of empirical hypotheses are established that relate (independent) attributes of some entities (e.g., software components) to other (dependent) attributes of the same or different entities. Dependent attributes are usually external quality attributes of software systems or parts thereof, e.g., reliability, maintainability, effort. Independent attributes capture factors that are assumed to have a causal relationship with the dependent attribute. (Example: Consider the measurement goal to predict development effort. The empirical hypothesis says product size is positively related to effort.) This is similar to the principle that designing experiments requires the definition of precise, testable research goals before any measurement is taken or any empirical study is proposed [39]. Clear goals lead to the definition of clear dependent and independent attributes, as they are referred to in software measurement
2. *Definition of measures for the independent attributes* (refined in Figure 3(b) and illustrated in Section 5). Independent attributes are formalized via sets of generic properties that characterize their measures (e.g., mathematical properties such as additivity). Entities are formalized via abstractions (e.g., dependency graphs) which are chosen based on the entities, the independent attributes and the generic properties. The generic properties are then instantiated on the abstractions and refined to take into account additional assumptions that depend on the specific application environment. Based on this refined set of properties, measures are defined or selected for the attributes of entities. Additional checks may be required to verify whether the defined measures really comply with the refined and generic properties. (Example: Consider size as measured by lines of code.)

3. *Definition of measures for the dependent attributes* (refined in Figure 3(c) and illustrated in Section 6). The GQM/MEDEA approach deals with independent and dependent attributes of entities in much the same way. (Example: Consider effort as measured in staff months.) In the context of experimental design, steps 2 and 3 correspond to the step of defining measurement procedures that optimize what is referred to as *construct* validity [39], 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.
4. *Hypothesis refinement and verification* (refined in Figure 3(d) and illustrated in Section 7). The empirical hypotheses are verified using the measures defined for the independent and dependent attributes. They may be refined by providing a specific functional form for the relationship between independent and dependent measures, e.g., an exponential relationship. These new, more specific empirical hypotheses are then used to build the predictive model based on actual development data. This model can be used to (1) verify the plausibility of empirical hypotheses and (2) as a prediction model. (Example: Consider the hypothesis that the number of staff months is a curvilinear function of lines of code. Consider analyzing staff month vs. lines of code data to build the functional relationship. This can be used to verify the specific curvilinear relationship and predict the staff months on the project of interest.) The definition of such a model will strongly influence the likelihood of obtaining significant results. For example, it may be important to take explicitly into account the presence of interactions among independent variables through, for example, the specification of multiplicative terms in a regression equation [32]. Typically, additional data analysis problems have to be addressed such as outlier analysis [1] or the statistical power [14] of the study. Briand and Wuest [12] provide detailed guidelines on these matters and an empirical validation procedure for software product measures.

Figures 2 & 3 show that most of the outputs (e.g., abstractions, measures) of the steps defined above are reusable. They should be packaged and stored so that they can be efficiently and effectively reused, thus reducing the cost of measurement in an organization [5]. 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.

To further formalize the concepts involved in the GQM/MEDEA process, we provide in Figure 4 a UML class diagram. The diagram can be seen as a starting point for the object-oriented design of a tool supporting the methodology and the reuse of measurement program information.

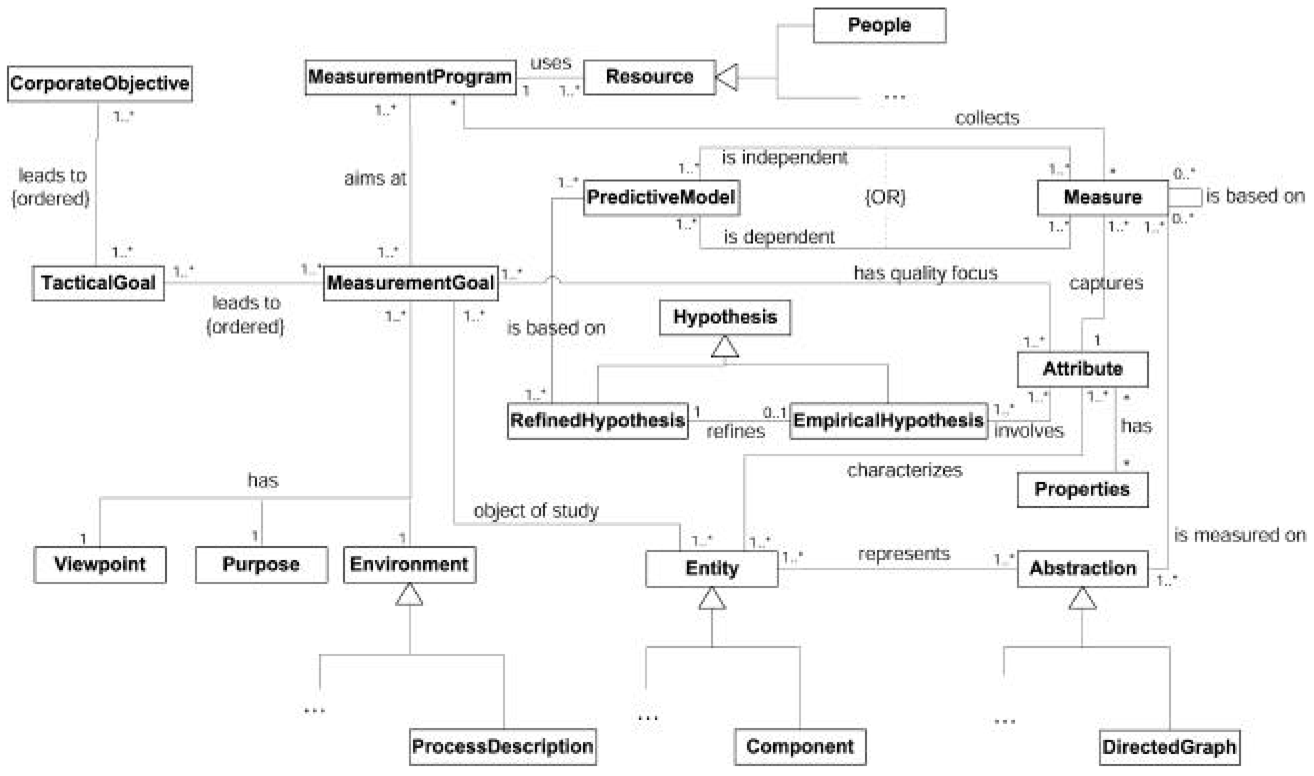


Figure 4. Conceptual Model Using a UML Class Diagram

In Figure 4, we see that TacticalGoals relating to a Corporateobjective are {ordered} as are MeasurementGoals relating to TacticalGoals, showing that such goals are prioritized. MeasurementGoals have a Viewpoint, Purpose, and Environment, as described by the GQM goal template. The quality focus is modeled by an association of the same name between MeasurementGoal and Attribute. TacticalGoal and MeasurementGoal are not expected to share associations and attributes and this is why no Goal superclass is defined.

A MeasurementGoal entails the measurement of Attributes, which are then associated with Measures. Attributes have Properties and are involved in Empirical hypotheses (EmpiricalHypothesis) describing relationships among two or more Attributes. A MeasurementGoal is part of a MeasurementProgram which consumes Resources, e.g., People. Measures are involved in PredictiveModels either as independent or dependent variables (modeled by a {OR} constraint). Measures are taken on Abstractions (e.g., DirectedGraph) which are analyzable representations of Entities (e.g., Component). Measures can be defined based on other Measures, hence the reflexive association on the class. PredictiveModels are defined based on refined hypotheses (RefinedHypothesis), which refine previously defined empirical hypotheses.

GQM filled the gap between goals and measures by means of questions and models [13], and gave recommendations on how to define them. GQM/MEDEA does not rely on questions to fill that gap, but it defines the steps to carry out, the relationships among them, the sources of information, and shows the integrated use of hypotheses, abstractions, and theoretical and empirical validation.

3. Two Applications

First, we concisely describe the context information for the two applications that we will use as case studies to illustrate our approach. Other information about the applications (e.g., characteristics of the languages used, goals, empirical hypotheses, etc.) will be provided in the description of the specific steps for which it is relevant.

Like many other software engineering empirical studies, our case studies may be classified as correlational, field studies.

Case Study 1
We studied three software systems developed at the Software Engineering Laboratory. The first system studied (GOADA) is an attitude ground support software for satellites developed at the NASA Goddard Space Flight Center (GSFC). The second one (GOESIM) is a dynamic simulator for a geostationary environmental satellite. These systems are composed of 525 and 676 Ada units, 90 Klocs and 170 Klocs, respectively, and have a fairly small reuse rate (around 5% of the source code lines have been reused from other systems, verbatim or slightly modified). The third system we studied (TONS) is an onboard satellite navigation system, which has been developed in the same environment and is about 180 Ada units and 50 Klocs large, with an extremely small rate of reuse (2% of the source code lines have been reused from other systems, verbatim or slightly modified). We selected projects with lower rates of reuse in order to make our analysis of design factors more straightforward by removing what we think is a major source of noise in this context. Additional information about this study can be found in [19].

Case Study 2
The system was developed within the framework of the ESSI project ELSA, which involved Politecnico di Milano, TXT and ENEL. The application domain was that of embedded systems for real-time applications for hydroelectric power plant control. Our study focused on the specification phase. The specification produced contained 33 TRIO+ classes (an object-oriented formal specification language, see Section 4.1.2), developed by five people: all of them were already experienced in both the application domain and the specification language. The process essentially followed the waterfall model, and the overall project lasted 18 months. The application domain (hydroelectric plants) was not new for the corporation, but this was the first time in which software measurement was used. Additional information about this study can be found in [16].

Sections 4 – 7 detail the high-level steps of the GQM/MEDEA approach shown in Figure 2. For illustration purposes, each section contains three subsections:

- Description of the step;
- Examples taken from the two application cases;
- Practical guidelines and common issues.

4. Set-up of the Empirical Study

The definition of the measurement goals and empirical hypotheses are the fundamental phases, since all the other steps in our approach are affected by them. Therefore, extra care must be used when setting goals and empirical hypotheses. Accurate descriptions of the software development process and knowledge acquisition techniques [10] can be used to better understand the issues that are most relevant and problematic in a given organization.

4.1. Define Measurement Goals

Description

Measurement goals are defined based on the general corporate objectives (e.g. reduce cycle time), the available information about the development environment (e.g., weaknesses, problems), and the resources available to carry out the empirical study.

The corporate objectives are of a strategic nature, in that they provide the company's general business directions for software development. Corporate objectives need to be prioritized, for instance based on the nature of the business domain. For instance, reduction of time-to-market may be more important than product quality in a specific business domain, while the converse may be true in other business domains. The measurement activities should be carried out in the context of the most important corporate objectives. The first, obvious reason is that the measurement program would bring a higher payoff to the corporation. The second reason is that the measurement program is more likely to be successful, since it is more likely to receive adequate support.

Information about the development environment, e.g., a process assessment results, can point to specific problem areas, whether processes or products, that need to be addressed. The corporate objectives and the information about the specific environment lead to the generation of tactical goals, which are narrower and better focused than corporate objectives, to whose achievement they contribute. For instance, in the context of a corporate objective such as "reduce development cost," a first tactical goal may be to "monitor testing effort" if the information about the specific environment points out problems in the testing phase effort. Several tactical goals may be established for the same corporate objective, and they should be prioritized before defining measurement goals, based on their importance in the context of the corporate objectives and available resources.

The tactical goals and the knowledge of the environment then lead to the establishment of measurement goals. Again, the measurement goals should be ranked based on their perceived relevance to the tactical goals, the resources they entail, and their feasibility with respect to building a useful prediction model (e.g., enough observations can be collected).

When proceeding from corporate objectives to tactical goals and to measurement goals, one should take into account the resources available for measurement, e.g., the number and skills of people involved, the tools that can be used to analyze software artifacts, and the resources of the software organization, which may not allow for certain kinds of empirical studies. For instance, it is unlikely that a software company will develop the same software in two different ways so that two different software design methods can be studied and compared.

This step requires a careful pre-study, e.g., process analysis and assessment, before measurement-related activities start. The information needed for defining measurement goals does not necessarily have to be quantitative in nature. However, it has to be clear and detailed enough so that one can establish sensible (i.e., relevant and attainable) goals and then interpret the experimental results. The existing documentation on the corporation and environment may be a useful starting point, especially if complemented with knowledge provided by the people involved in the development environment.

Performing this step in a systematic manner requires goal definition techniques. We use the Goal/Question/Metric (GQM) paradigm [13, 2, 5, 6], which provides a template and guidelines to define measurement goals and refine them into concrete and realistic questions, which subsequently lead to the definition of measures. Here is a summary of templates that can be used to define goals:

Object of study: products, processes, resources, ...
Purpose: characterization, evaluation, prediction, improvement, ...

***Quality focus**:* cost, correctness, defect removal, changes, reliability, ...
***Viewpoint**:* user, customer, manager, developer, corporation, ...
***Environment**:* team, project, product, ...

These five goal dimensions have a direct impact on the remaining steps of the measure definition approach and the data collection program.

The ***object of study*** helps determine the

- *entities* that must be modeled by abstractions so as to be analyzable, e.g., a UML class diagram [41] for a program design;
- *hypotheses* that may be relevant because they are related to the object of study, e.g., reducing coupling among classes improves maintainability.

The object of study might not completely specify the set of entities that need to be studied, since

- other entities may need to be studied as well (e.g., entities whose attributes are believed to significantly influence the attribute of the object of study that appears in the quality focus, such as the designers' experience when studying a UML class diagram);
- the object of study may need to be studied at a finer granularity level than that mentioned in the GQM goal template (e.g., if the object of study is a software system, we may want to study the modules of which it is composed).

The ***purpose*** shows the intended use of the measures to be defined. Though we here concentrate on prediction systems, several measurement process characteristics are in general affected by the purpose dimension of a GQM goal. For the sake of discussion, we compare how these characteristics are affected by the prediction purpose and the characterization purpose (i.e., understanding the typical distributions of data in a given environment [5, 13]). The purpose helps determine the

- *type of data to be collected*, e.g., as opposed to characterization, prediction requires data that can be accurately collected or estimated at the time of prediction;
- *amount of data to be collected*, e.g., prediction usually requires more data than characterization so that possibly non-linear, complex relationships have a chance to be statistically detected;
- *need for empirical validation*, e.g., if the measurement model is predictive, as is our case, empirical validation of the assumptions (e.g., functional shape of the prediction model) is necessary, but not if the purpose is characterization.

The ***quality focus*** helps determine the

- *dependent attribute* used in the hypotheses, e.g., development cost;
- *hypotheses* describing the relationship between the attributes of the object of study and the dependent attribute, e.g., cost is related to system size.

The ***viewpoint*** helps determine

- *the point in time* at which predictions should be carried out and therefore what information will be available to define abstractions and measures;
- *the definition of descriptive models for the quality focus [13], that is the relevant dependent attributes*: for example, from the user's point of view, reliability may be defined as the mean time to failure, whereas, from the tester's point of view, it may be defined as the number of faults detected over a period of time;
- *what information is costly or difficult to acquire* and, consequently, what information should possibly be left out of the model.

The ***environment*** helps determine the

- *context* in which the study is carried out;
- *scope* in which the results of the study are valid and can be generalized.

As the measurement goal definition phase is the most important one, since it influences all other steps, it is not surprising that one may return to it from the other step of the "Setting of the Empirical Study" phase. One may need to fine-tune the variables of the GQM goal template, based on the empirical hypotheses defined. For instance, this is what happened in our Case Study 1 (Section 4.2).

Examples

We now illustrate the description above with our two case studies.

Case Study 1
We focused on one general corporate objective:

- reduce the number of defects in the end product.

Given the application domain of GSFC, product quality was the highest priority for software development. This is a very broad objective, in that it may cover the entire software development process. Based on information on the process and evidence on the projects, our tactical goal required that, among the several process- or product-related entities (e.g., process phases or activities, or artifacts) that were in principle worth studying in our environment, we study the high-level design, which was believed to greatly influence the final product's quality, as also commonly believed in software engineering. Studying the high-level design was believed to be feasible with our resources and constraints; studying other entities that we could presume were as influential as high-level design would have required more resources and case studies that were difficult to undertake.

In our application, we model the high-level design of a software system as a collection of Ada83 interfaces of packages and procedures (which we will call Ada modules), related to each other via

- USES relationships (e.g., package/procedure *A* USES package *B* if *A* uses computational services that package/procedure *B* makes available), and
- IS_COMPONENT_OF relationships (e.g., package/procedure *A* IS_COMPONENT_OF package/procedure *B* if *A* is defined within *B*).

Precise and formalized information on Ada module bodies is not available this early in the lifecycle.

Ada modules belong to higher level aggregations, represented as library module hierarchies. In a library module hierarchy, nodes represent Ada modules, arcs between nodes represent IS_COMPONENT_OF relationships, and there is exactly one top level node, which is a package. In this paper, we define attributes and measures that can be applied to both modules and library module hierarchies, which are the most significant syntactic aggregation levels below the subsystem level. We use the term *software part* (*sp*) to denote either a module or a library module hierarchy.

Our measurement goal was defined as:

Goal 1.1
Object of study: high-level design of Ada software systems
Purpose: prediction
Quality focus: fault-proneness of the implemented systems
Viewpoint: project leader and development team
Environment: software projects at the GSFC Flight Dynamics division

After the empirical hypotheses were made explicit, Goal 1.1 was modified (see Section 4.2.2).

Case Study 2
Among others, two general corporate objectives were deemed important in this context:

- reduce the number of defects in the end product;
- plan software development better.

The first corporate objective is the most important one because of the application domain, where failures may be disastrous. The second corporate objective, though not as important as the first one, was studied because the empirical study was seen as a good opportunity for the software organization to start acquiring quantitative information on which to base software development resource planning. Quantitative information on this specific type of applications (and on projects where formal methods are used) was not available.

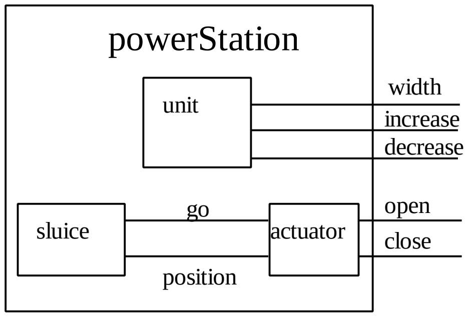
We restricted our tactical goal to studying the above objectives in the specification phase, which is carried out in time-critical systems with greater care than in other systems, so it was by far the most important one in the development process. Also, the design and implementation phases were quite straightforward once the TRIO+ specification was available because of the formal nature of TRIO+ and the fact that many TRIO+ clauses of the specification, whose form was precondition$\rightarrow$postcondition, afforded direct translation into software code, in the form "**if** precondition **then** action to achieve the postcondition"). These two GQM goals were identified.

Goal 2.1
Object of study: TRIO+ specification
Purpose: prediction
Quality focus: effort required to write the specification
Viewpoint: project leader
Environment: development of software for hydroelectric plants

Goal 2.2
Object of study: TRIO+ specification
Purpose: prediction
Quality focus: changes in the specification
Viewpoint: project leader
Environment: development of software for hydroelectric plants

Identifying the factors that contribute to effort and changes may allow the corporation to carry out appropriate actions to reduce both and help plan specification effort and stability.

The object of study for both goals is a TRIO+ specification. To make the paper self-contained, we now provide a brief introduction to the TRIO+ specification language. TRIO+ is an object-oriented formal specification language based on temporal logic that can be used to capture time-related specifications along with functional specifications in an integrated manner. This characteristic is of crucial importance for describing real-time systems. TRIO+ also satisfies several expressive requirements relevant to large systems, such as possibility of encapsulation, information hiding, existence of different levels of abstraction, etc., which are fundamental for the building of specifications of large systems. Figure 5 (taken from [35]) shows a simple example of a TRIO+ specification for a power station in both graphical and textual form.



```
class   powerStation
visible   width,open,close,increase,decrease
temporal domain    real
TD items
          vars  width: real
          predicates    increase,decrease,open,close
module  unit: powerUnit
          sluice: sluiceGate
          actuator: gateActuator
connections          {(actuator.go sluice.go)
                      (sluice.position actuator.position)
                      (unit.width width)
                      (increase unit.increase)
                      (decrease unit.decrease)
                      (open actuator.open)
                      (close actuator.close)}
end  powerStation
```

Graphical description

Textual description

Figure 5. A TRIO+ specification.

Here is a short summary of the TRIO+ definitions relevant to this paper:

Class is an encapsulation mechanism that allows information hiding. A class may contain instances of other classes, called modules. In the example of Figure 5, class *powerStation* contains (clause **module**) modules *unit, sluice, actuator.*

Items are predicates, variables, functions, and constants. Local items are only available to the class where there are defined; visible items are also available to other classes. In Figure 5, items *width*, *open*, *close*, *increase*, *decrease* are visible items for class *powerStation* (clause **visible**); Items *actuator.go*, *sluice.go*, etc. are visible items for the modules, but local items for *powerStation*.

Connections state that two items belonging to different classes are actually the same item. As an example, there is a connection between *actuator.go* and *sluice.go*.

Axioms are temporal logic expressions stating properties that must hold for a TRIO+ class. For instance, an axiom (describing the opening of a gate) for class *sluiceGate* is

move_up: position =mvUp **and** go(down) → ∃t(NextTime(position=up,t)
and Futr(Lasts(position=mvDown,Δ) **and** Futr(position=down,Δ,t))

where

- *move_up* is the name of the axiom,
- *NextTime*, *Futr*, and *Lasts* are operators provided by the TRIO language [35],
- *up, mvUp*, *down*, and *mvDown* are possible values for *position*,
- *D* is a constant.

Discussion

The use of a goal-oriented paradigm such as the GQM paradigm provides two important results:

- The data are collected for a purpose, so they are ensured to respond to the specific needs of the software organization;
- The derivation of measures from explicit goals allows the analyst to specify *a priori* the interpretation mechanisms associated with the collected data.

To illustrate the impact of measurement goals in our approach, take Goal 1.1 as an example. We know from the *object of study* that we have to define relevant mathematical abstractions for high-level design. We know from the *purpose* of measurement that we need to collect enough data about the quality focus to allow for a statistically significant validation of the relationships between the measures of attributes of the object of study and quality focus. This requires that we better define our *quality focus*: fault-proneness. Very likely, we need to determine precisely how to count defects, e.g., what testing and inspection phases should be taken into account? are all faults equal or should they be weighted according to a predefined fault taxonomy? Such questions are also dependent on the particular *viewpoint*. In our example, the project leader and the development team want to find out where faults are located. In other cases, the project leader and the development team might be particularly interested in critical faults (according to their own definition of criticality). Therefore, faults will have to be "weighted" according to the level of criticality of their consequences. The *environment* shows the (minimal) scope in which the results of the empirical study are valid, i.e., software projects in the flight dynamics division of GSFC (as explained in Section 4.2.1, this dimension was changed to low-reuse projects in the flight dynamics division of GSFC). The results may be valid in other environments, but a careful study of their similarities and differences with the original environment should be carried out first.

4.2. Define Empirical Hypotheses

Description

We state hypotheses on aspects of the software process that are relevant to the experimental goals. An *empirical hypothesis* is a statement believed to be true about the relationship between one or more attributes of the *object of study* and the *quality focus*. A hypothesis captures one's own intuitive understanding of the studied phenomena and needs to be explicit so it can be discussed, questioned, and refined. Various sources of information can be very useful for devising pertinent hypotheses, such as a thorough understanding of the working procedures, methodologies, and techniques used in the development environment (i.e. process description) as well as product information and domain experts' knowledge (e.g., obtained through interviews) [10].

At this point, several different empirical hypotheses may be defined. The verification of empirical hypotheses will show which hypotheses are plausible (in a statistical sense).

Examples

We provide a few examples of hypotheses taken from the two case studies. Other hypotheses were defined as well, but not all of them were supported by empirical evidence (see Section 8).

Case Study 1
Hypothesis 1 (coupling and fault-proneness):
The larger the import coupling of a software part, the larger its fault-proneness. ("Import" coupling refers to the dependence of a software part on other software parts.)

Hypothesis 1 states a common belief in software engineering that was also considered true in our application context; it is widely accepted that a software system should be composed of modules with a small degree of coupling.

The study of this empirical hypothesis required that we identify and remove possible factors (confounding factors) that could mask or inflate the effect of coupling on fault-proneness. We identified a high level of reuse as a potential confounding factor, so we narrowed the environment of our study to low-reuse projects (< 5%). Thus, we went back to the previous phase and modified the environment variable to "low-reuse software projects at GSFC." The GQM goal becomes

Goal 1.1
Object of study: high-level design of Ada software systems

Purpose: prediction
Quality focus: fault-proneness of the implemented systems
Viewpoint: project leader and development team
Environment: low-reuse software projects at GSFC

As a part of the “Define Measurement Goals” step, we checked the feasibility of the goal, by checking if there was a sufficient statistical basis for the validation of the empirical hypothesis in the new context. As this was the case, we proceeded to execute the subsequent phases.

Case Study 2
Hypothesis 2 (class axiom size and effort):
The larger the size of a TRIO+ class in terms of axioms, the higher the effort required to write it.

The basic idea is that each additional TRIO+ axiom requires additional effort to be written and additional effort to check its consistency with the other axioms.

Hypothesis 3 (class axiom size and changes):
The larger the size of a TRIO+ class in terms of axioms, the higher the amount of changes in it.

The idea of the hypothesis is that each additional TRIO+ axiom is likely to cause more inconsistencies.

Discussion

The above hypotheses may be deemed too simple; for instance, they do not describe a specific functional correspondence between two variables. However, this is not possible at this point in the measure definition process, since the empirical hypotheses involve attributes of entities, and not measures. In the *Instantiate and Refine Empirical Hypotheses* step, more precise hypotheses may be made after measures for the independent and dependent attributes have been defined.

Empirical hypotheses help identify the measurement attributes (e.g., size, complexity, cohesion, coupling, likelihood) that are believed to be relevant to the goal. In general, a hypothesis may involve several independent attributes and one dependent attribute. Also, empirical hypotheses allow us to better identify artifacts, activities, or parts thereof that must be taken into account for the definition of suitable abstractions.

The empirical hypotheses we state are different from those that are defined in statistical test of hypotheses. First, our empirical hypotheses are defined in terms of attributes, while statistical hypotheses are defined in terms of measures. Second, statistical test of hypotheses requires that two exhaustive and mutually exclusive hypotheses be tested, the null hypothesis and the alternative hypothesis, which is usually the hypothesis that the experimenter would like to support through the experiment. Our empirical hypotheses are akin to the alternative hypotheses and the counterpart of the null hypothesis of statistical test of hypothesis is not made explicit here, but it is the logical negation of the empirical hypothesis stated. Our empirical hypotheses eventually lead to the statement of statistical hypotheses that are subject to statistical testing (in Step 4). At any rate, it is important to describe the empirical hypotheses as precisely and unambiguously as possible. This facilitates the process for defining measures and the interpretation of results.

5. Definition of Measures for Independent Attributes

Based on the entities and attributes appearing in the empirical hypotheses and on process and product information, a new measure is defined or an existing one is selected as the result of this phase. This measure should

1. comply with a set of properties that are believed to be true for all measures of the attribute it purports to measure;
2. comply with a set of additional properties believed to be true in the environment under study;
3. be measured based on a suitable mathematical abstraction of the object of study.

5.1. Formalize Independent Attributes

Description

The independent and dependent attributes of entities that appear in the hypotheses must be clearly identified, so the characteristics of their measures can be formalized to the extent possible. To describe the characteristics of measures for a given attribute, we use mathematical properties, which can be used to (1) constrain and guide the search for new measures for the attribute, and (2) accept existing measures as adequate for that attribute. One should always make sure that a measure exhibits properties that are essential for its technical soundness. The properties that we use are independent of both any specific abstraction and any instantiation of the attribute into any specific measure, so they are called *generic*.

The generic properties for the measures of an attribute should be logically consistent. Also, these properties should hold for the admissible transformations [26] of the level of measurement (i.e., nominal, ordinal, interval, ratio, absolute) on which one intends to define measures. In other words, there should not be any contradiction between the level of measurement assumed while using/interpreting a measure and its generic properties. The choice of the level of measurement must be based on the precision of the results that one would like to obtain, i.e., on the measurement goals and application needs. Thus, the choice of a measurement level influences the kind of refined empirical hypotheses one may state in the *Instantiate and Refine Empirical Hypotheses* step. For instance, the investigation of a refined empirical hypothesis that states a *linear correlation* between an independent variable (i.e., a measure for the independent attribute) and a dependent variable (i.e., a measure for the dependent attribute) requires that both measures be defined on at least an interval level of measurement[1]. Instead, the *association* between two variables[2] can be studied even if the two variables are measures defined on an ordinal level of measurement. A prediction made through an association is less informative than one carried out via a linear correlation, but it is less demanding in terms of the hypotheses on the underlying measure definition and precision.

Examples

We provide properties that are, in our opinion, generic for size measures [18][3]. Our properties are defined based on an abstract graph-theoretic model, since we may want to use them on all artifacts produced during software development. We do not intend to define sets of generic properties that should hold for single-valued measures that capture *all* aspects of size. Like in [26], we do not believe that such measures exist. On the contrary, our sets of generic properties should hold for a set of measures that address a specific aspect of size.

Size is an attribute of a system, i.e., one can speak about the size of a system. In our general framework—we want these properties to be as independent as possible from any specific abstraction—, a system is characterized by its elements and the relationships between them.

---

[1] Though the problem is not as clear cut, as discussed in [15].

[2] Computed using statistics such as Kendall's Tau or Spearman Rank Correlation Coefficient.

[3]For brevity, we do not show the properties of [18] related to coupling. Also, the reader is referred to [18] for a thorough discussion of these properties and their comparison with those existing in the literature.

Definition 1: Representation of Systems and Modules
A *system* S is represented as a pair <E,R>, where E represents the set of elements of S, and R is a binary relation on E ($R \subseteq E \times E$) representing the relationships between S's elements. Given a system S = <E,R>, a system m = <E_m,R_m> is a *module* of S if and only if $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$, and $R_m \subseteq R$. This is denoted by $m \subseteq S$.
Size cannot be negative (property Size.1), and we expect it to be null when a system does not contain any elements (property Size.2). When modules do not have elements in common, we expect size to be additive (property Size.3).

Definition 2: Size Measure
A measure of the size of a system S = <E,R> is a function Size(S) that is characterized by the following properties Size.1 - Size.3.

Property *Size.1*. Non-negativity:
$\text{Size}(S) \geq 0$ (Size.I)

Property *Size.2*: Null Value
$E = \emptyset \rightarrow \text{Size}(S) = 0$ (Size.II)

Property *Size.3*: Module Additivity
$(m_1 \subseteq S$ **and** $m_2 \subseteq S$ **and** $E = E_{m1} \cup E_{m2}$ **and** $E_{m1} \cap E_{m2} = \emptyset)$
$\rightarrow \text{Size}(S) = \text{Size}(m_1) + \text{Size}(m_2)$ (Size.III)

Properties Size.1-Size.3 are meaningful only for ratio measures [26].

Discussion

Even though there might be wide consensus on some of the above properties, acceptance of a set of properties for an attribute is ultimately a subjective matter, like for any formalization of an informal concept. As a matter of convenience, a universal set of properties should be defined for the most important attributes used by the software engineering community, as is the case for more mature disciplines. However, software engineering has not yet reached a satisfactory definition of the properties associated with most attributes. Therefore, it is important that all properties be explicit and justified so that their limitations may be understood and the discussion on their validity may be facilitated. A wide consensus can only be reached through the discussion and refinement of existing sets of properties. For instance, different sets of properties for complexity measures [42, 40] (though these are defined with reference to software code) may be used as an alternative to the set of properties defined in [18]. Thus, during the definition of measures, several choices are possible, based on the application at hand and one's own intuition.

A set of properties like the ones above should be interpreted as necessary, but not sufficient. Not all the measures that satisfy a set of properties associated with an attribute can be considered sensible measures for that attribute. This is the case even for the sets of properties that have long been accepted, such as the set of properties for distance. However, those measures that do not satisfy the set of properties associated with an attribute can safely be excluded. In addition, an explicit definition of the set of properties provides a formal means to reason about the measures for an attribute.

Though the size properties we provided above are meaningful for ratio measures, not all the measures that satisfy them are ratio ones. If a measure M does not satisfy the set of properties for an attribute A that are meaningful at the ratio level of measurement, all that can be safely inferred is that M is not a ratio measure for A. Satisfaction of a set of properties meaningful at the ratio level of measurement can at best be interpreted as supporting evidence for a measure to be a ratio one. It is also possible to provide properties that are meaningful for measures at other levels of measurement, so the definition of measures need not be constrained to the definition of ratio scales only [34].

Other ways of formalizing attributes may be used such as Measurement Theory [26, 27]. As shown in [18], the use of properties is not in contradiction with Measurement Theory. In general, it is up to the person in charge of defining a measure to identify the appropriate level and way of formalization for the attribute he or she investigates.

5.2. Identify Abstractions for Measuring Independent Attributes

Description

An abstraction is a mathematical representation of an entity, e.g., a graph. An entity may be mapped into one or more abstractions so it becomes analyzable and its relevant attributes become quantifiable [33]. The choice of an abstraction should be guided by the attributes, their formalization, and the empirical hypotheses, since the abstractions must help adequately capture the independent attributes of the entities involved in the hypotheses. The mapping from the entity to the abstraction needs to be checked for completeness, e.g., does the abstraction contain all the relationships between nodes that one wants to capture? Is the level of granularity of the abstraction nodes sufficient to represent the entity accurately? One way of assessing the suitability of an abstraction is to study the effect of relevant modifications on the entity (e.g., product) and assess its impact on the abstraction, e.g., number of nodes and edges added or removed, change of topology in a graph. Abstractions capturing control flow, data flow and data dependency information are available in the literature [36, 8, 37], and a large variety of abstractions can be derived from software products.

Examples

Case Study 1
The entities to be studied are software parts, and the attribute to be studied is coupling. Therefore, we are interested in capturing coupling in an object-based context, rather than a procedural one. We focus on the dependencies among data declarations (i.e., types, variables, or constants) and procedures. These dependencies are called *interactions* and are used to define measures capturing coupling between software parts. For instance, a data declaration *A* interacts with another data declaration *B* if *A* appears in *B's* declaration or in the right hand-side of an assignment in which *B* appears on the left-hand side. A data declaration *A* interacts with a procedure *C* if *A* interacts with at least one of *C's* data declarations (e.g., with one of *C's* formal parameters). The interaction relationship between data declarations is transitive. (See [19] for more detailed definitions.) Of the four possible kinds of interactions (from data declarations to data declarations; from data declarations to procedures; from procedures to procedures; from procedures to data declarations), we take into account only those interactions (1) from data declarations to data declarations or (2) from data declarations to procedures that can be detected from the high-level design of a software system. The other two kinds of interactions are not detectable at high-level design time.

The abstraction we use to model the elements and relationships that are relevant for capturing coupling is the interaction graph: its elements are the data declarations of the high-level design (including procedures' formal parameters) and its relationships the data declarations interactions. For instance, Figure 6(b) contains the interaction graph for the Ada fragment in Figure 6(a).

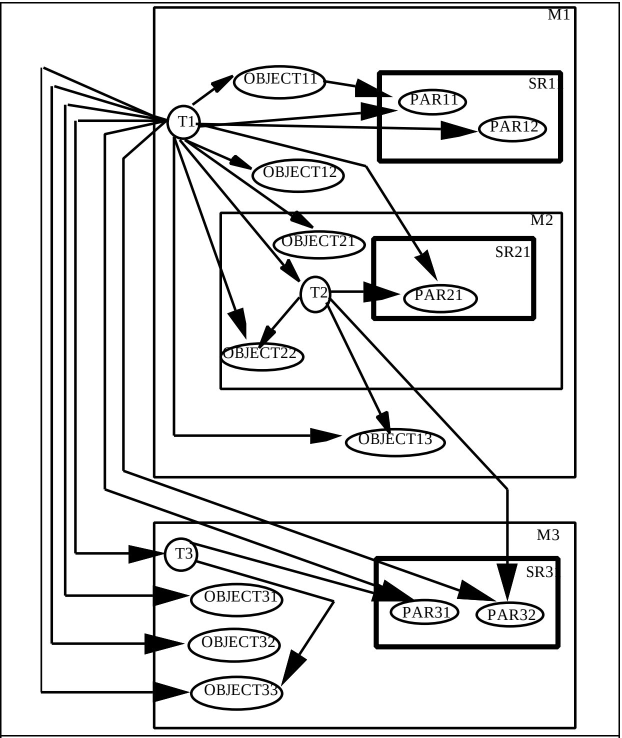
Case Study 2
To measure the size of TRIO+ classes in terms of axioms, we used a simple abstraction. Each axiom in a TRIO+ class is seen as an element of the system. Axioms in the same class are linked by means of

sequence relationships (like lines of code in programs). Axioms belonging to different classes are not linked to each other.

```
package M1 is
    …
    type T1 is …;
    OBJECT11, OBJECT12: T1:= …;
    procedure SR11(PAR11: in T1:=OBJECT11, PAR12: in T1);
    …
    package M2 is
            …
            OBJECT21: T1;
            type T2 is array (1..100) of T1;
            OBJECT22: T2;
            procedure SR21(PAR21: in out T2);
            …
    end M2;
    …
    OBJECT13: M2.T2;
    …
end M1;


with M1; use M1;
package M3 is
    …
    type T3 is array (1..100) of T1;
    OBJECT31, OBJECT32: T1;
    procedure SR31(PAR31: in T3, PAR32: in M2.T2);
    OBJECT33: T3;
    …
end M3;
```

(a)



(b)

Figure 6. Ada-like code fragment (part a) and its interaction graph (part b)

5.3. Instantiate and Refine Properties (for Measures of Independent Attributes)

Description

The elements, relationships, and modules of abstractions must be mapped onto the elements, relationships, and modules of systems and modular systems. This correspondence is sometimes straightforward. However, it must be made explicit, at least in principle, to make sure that there are no hidden problems in the transition from one step to the other. Then, the set of properties associated with each attribute is expanded by adding new properties, which formalize additional knowledge on the characteristics of the measures for that attribute in the specific context and allow us to tailor the generic measurement attributes to any particular quality focus. These properties are believed to be true in a given context of measurement (i.e., goals, attributes, empirical hypotheses, abstractions) and are referred to as *context-dependent properties*. These additional properties are often left implicit during measure definition.

Examples

Case Study 1

The elements, relationships, and Ada modules of an interaction graph are mapped onto elements, relationships, and modules of a modular system.

Context-Dependent Property 1: Coupling interactions (based on hypothesis 1)

Given two software parts *sp1* and *sp2*, if *sp1* has at least as many coupling interactions as *sp2*,

$\text{Coupling}_{\text{CouplingInteractions}}(\text{sp1}) \geq \text{Coupling}_{\text{CouplingInteractions}}(\text{sp2})$.

$\text{Coupling}_{\text{CouplingInteractions}}$ represents a measure that captures the coupling of a software part based on coupling interactions.

Case Study 2
Axioms are mapped onto elements of a system. Sequence relationships of axioms belonging to the same class are mapped onto relationships of the system.

Context-Dependent Property 2: Axioms (based on hypotheses 2 and 3)
If a class C1 has at least as many axioms as another class C2, then $\text{Size}_{\text{Axioms}}(\text{C1}) \geq \text{Size}_{\text{Axioms}}(\text{C2})$.

Discussion

The context-dependent properties above are not implied by the generic properties for coupling and size discussed above. For instance, Context-Dependent Property 2 is an additional property, which basically implies that all axioms should have the same "weight" as for the facet of size we are interested in. Context-dependent properties must be made explicit, though they are often kept implicit in articles that propose new measures. Hypotheses and, consequently, context-dependent properties, may be questioned and refined in later research. For instance, based on the experimental results obtained with the above empirical and context-dependent properties, one may draw the conclusion that axioms should be given different "weights," depending on the number of predicate occurrences in an axiom. This has the following consequences:

1. there exists a different hypothesis, in which the occurrences of predicates in axioms are considered relevant, instead of the set of axioms in classes;
2. a new abstraction needs to be built, where predicate occurrences are the elements;
3. a new context-dependent property must be provided, e.g., to rank classes whose axioms differ by the number of predicate occurrences;
4. new measures are defined;
5. a new empirical validation is required.

Therefore, a "small" change in terms of hypotheses and context-dependent properties can have a dramatic influence over the whole measure definition process.

One of the main difficulties of this step is to ensure that the set of context-dependent properties is complete, i.e., any pair of abstractions can be ordered by using the stated properties. Sets of generic properties do not necessarily provide a total order among entities, as not all abstractions may be comparable with respect to a particular measurement attribute [25, 24], so only a partial order can be obtained [33]. Context-dependent properties further constrain the order relation among entities in such a way that a total order can be obtained, though this is not true in general. For instance, a total ordering of the entities may not be obtained when dealing with multidimensional attributes, or several aspects of the same attribute (e.g., data flow and control flow complexity, or even control flow complexity alone, as shown in [24]).

5.4. Define Independent Measures

Description

For each attribute of an entity, measures are defined by using the abstractions' elements and relationships, and are checked against the attribute's generic and context-dependent properties.

Management and resource constraints are taken into account for defining convenient measures. This step may require approximations which must be performed explicitly, based on a solid theory, and in a controlled manner.

Examples

Here, we report a few of the measures we identified in our application cases.

Case Study 1
Two simple coupling measures are obtained by counting the number of input interactions of a software part. In this case, one may choose whether to consider all such interactions (measure TIC) or only a subset (measure DIC).

Measure TIC for Import Coupling
Given a software part *sp*, Transitive Import Coupling of *sp* (denoted by TIC(sp)) is the number of interactions between data declarations external to *sp* and the data declarations within *sp*.

Measure DIC for Import Coupling
Given a software part *sp*, Direct Import coupling of *sp* (denoted by DIC(sp)) is the number of direct interactions, i.e., those that are not obtained only through transitivity of interactions.

Case Study 2
A simple axiom-based size measure is defined as follows.

Measure Axioms for Axiom Size
Axioms is the number of axioms of a class.

Discussion

At this stage, we may not able to select the best among alternative measures satisfying generic and context-dependent properties. Empirical validation (Section 7.2) may help perform such a selection, e.g., by identifying the best predictor. As a necessary precondition to carrying out a meaningful experimental validation, the level of measurement of the measures must be determined. This prevents measures from being misused (e.g., taking the average value of an ordinal measure, which is meaningless from a measurement theory standpoint).

5.5 Validate Independent Measures

Once measures have been defined, it must be proven that they are consistent with the generic and context-dependent properties.

Description

The measures for the independent attributes may not satisfy the refined properties for the independent attributes. The measure definition process—like any human-intensive activity—is subject to errors, and cannot provide 100% certainty that correct results are always delivered. It is useful to make sure that the measures for the independent attributes comply with one's formalized intuition. In addition, before reusing existing measures, one should check if they satisfy the refined properties for the attributes they purport to measure.

Examples

It can be shown that the measures defined in the Define Independent Measures step for both application cases satisfy the refined properties for the attributes they purport to measure. For the sake of brevity, we refer the reader to [16, 19].

Discussion

The activity of this step can be useful in two ways:

1. filter out and discard measures that do not comply with one's intuition; execution of the process may then resume from one of the previous steps, if necessary;
2. identify the extent to which measures agree with one's own intuition; measures that do not entirely satisfy refined properties might not be discarded, but one has a better idea of the strengths and weaknesses of measures and their degree of approximation of the attributes they try to measure.

6. Definition of Measures for Dependent Attributes

The definition of measures for the dependent attributes mirrors the activities shown in Section 5.

6.1 Formalize Dependent Attributes

Description

Dependent attributes are often better understood than independent attributes, since they are usually more tangible. For instance, an attribute such as cost is much more easily understood than code complexity on an intuitive level. Therefore, in most cases, the need for formalizing the properties of measures for dependent attributes is somewhat less felt than that for formalizing the properties of measures for independent attributes. However, this is the result of a longer acquaintance of researchers and practitioners with those attributes during their professional activities and in real life, and not always the consequence of a greater ease in defining dependent attributes. Also, even though not explicitly used, a formalization of the properties for these attributes may already be available, as we show next.

Examples

Case Study 1
The dependent attribute is fault-proneness of a software part, which may be interpreted as the probability to have at least one fault in a software part. We need to provide properties that characterize the measures for the probability attribute. Since probability has been studied and used for centuries, these properties are already available [28] and we can "reuse" them.

To introduce the properties for probability measures formally, we first need to introduce the basic elements upon which these properties are based. A random experiment may have several *outcomes*. The set of all outcomes of a random experiment is called the *sample space* of the random experiment. A set of outcomes (i.e., a subset of the sample space) is called an *event*. The sample space of a random experiments is the set of its possible outcomes. Therefore, the properties of probability measures are defined based on a very simple—and general—set-theoretic model. We now report the three properties that characterize probability measures [28].

Definition 3: Probability
A probability measure is a real valued set function defined on a sample space that satisfies the following three properties (*Probability* denotes any such measure).

Property *Probability.1: Admissible Range*
Given any subset A of the sample space S

$0 \leq \text{Probability}(A) \leq 1$

Property *Probability.2: Maximum value*

The probability measure of the whole sample space S is maximum
Probability(S) = 1

Property *Probability.3: Additivity*
The probability measure of the union of a finite or infinite collection of disjoint events A_1, A_2, … is the sum of the probabilities of the individual events
Probability($A_1 \cup A_2 \cup \ldots$) = Probability(A_1) + Probability(A_2) + …

Case Study 2
There are two dependent attributes, i.e., effort and amount of changes. They can both be interpreted as facets of size, whose properties have been described in Section 5.1.2.

Discussion

It is worth noting that properties for process attributes are usually left implicit by researchers and practitioners. This is not because of a lack of formalization or consensus, but exactly for the opposite reason: probability is a well-understood attribute, for whose meaning a wide consensus exists, so the properties of its measures are taken for granted. The goal of any formalization endeavor (such as those based on properties of [42, 18]) is to provide a ground for discussion and, through modifications and refinements, reach a wide consensus, as happened in the past for many other attributes for which a widespread agreement now exists.

6.2 Identify Abstractions for Measuring Dependent Attributes

Description

Abstractions must be found to capture the entities' relevant aspects for capturing our dependent attributes, which are referenced by the empirical hypotheses. The nature of these abstractions may not be as rigorously mathematical as that of the abstractions used for the entities of the independent attributes, especially if the dependent attributes refer to process entities such as phases or activities. However, the relevant aspects of the abstractions (e.g., phase milestones) need to be clearly identified, since the measures and the data collection process will be based on them.

Examples

Case Study 1
We need to study the final software systems (entities) with reference to the faults (aspects) they may contain. Thus, we need to clearly state what we mean by fault in our context, i.e., we need precise rules so we can unambiguously identify faults. For instance, we need to decide how to handle "multiple" faults, i.e., those defects that are replicated in the software code. Suppose that a variable is incorrectly used instead of another variable in several points of the software code. Does this represent a single fault? Or, do we need to count each incorrect use of that variable as an individual fault? Such a decision must be made explicit and must be consistently applied in the empirical study. In our application case, we used the definitions for faults that had been used at the GSFC Software Engineering Laboratory for a number of years.

Case Study 2
Two kinds of entities must be considered: the specification process, whose relevant attribute is effort, and the various versions of the specification, to identify the changes from one version to the other. In both cases, it was not deemed useful to model the specification process and the various versions of the specification in detail. Effort was simply modeled by recording the effort needed to write and revise the specification. As for changes, the analysis concentrated on massive modifications, or changes that

were deemed important by the specifiers and justified the building of a new version. Thus, the existence of a new version for a class in the TRIO+ specification was equated to a significant change.

Discussion

The decision of the level of detail with which an abstraction represents an entity depends on the expected results of the measurement activity. For instance, in a more thorough analysis, one might have decided to differentiate various categories of faults (in Case Study 1), or take into account the extent of changes (in Case Study 2). A greater level of detail has to be justified by the accuracy required for the predictive models and needs to be evaluated against the available resources.

6.3 Instantiate and Refine Properties (for Measures of Dependent Attributes)

Description

The generic properties for dependent attributes must be instantiated and may be refined. In this case, instantiation may be simpler than for independent attributes, due to the nature of dependent attributes. Also, refinement may not be required.

Examples

Case Study 1
The outcome of the random experiment is instantiated as the absence/presence of at least one fault in a software part.

Case Study 2
Since all specifiers had equivalent skills, it was reasonable not to make any differences as for the effort each of them spent in the specification phase. For simplicity, all changes (i.e., the "differences" between to consecutive versions of the same class) were considered of the same extent.

6.4 Define Dependent Measures

6.4.1 Description

Measures for dependent attributes look more straightforward than those for independent attributes. Rigor and care are still required, though problems may usually be solved by using common sense.

6.4.2 Examples

Case Study 1
The probability of an event (in our case, the presence of at least one fault) cannot be measured directly, but it must be estimated. Depending on the specific random experiment, different estimation mechanisms are used. For instance, one estimates the probability *p(e)* that some event *e* occurs by carrying out *n* trials and counting how many times *e* occurs. If the number of occurrences of *e* is denoted by *x*, the probability *p(e)* is estimated as $\hat{p}(e) = x/n$, i.e., the maximum likelihood estimate of *p(e)*. This shows that the actual measure of a probability is necessarily a derived measure (called indirect measure in [26]), and a formula must be provided for it. In our case, we used Logistic Regression, which is flexible enough to allow for the modeling of a number of different relationships. Logistic Regression is a classification technique for estimating the probability that an object belongs to a specific class, based on the values of the independent variables that quantify the object's attributes. The Logistic Regression equation shows that data on faults must be collected to evaluate fault-proneness. In this case, it is not required to count faults, but only to record them.

For explanation purposes, we here address the case of a dependent variable Y which can take only the two values 0 and 1, and any number of independent variables X_i. The multivariate Logistic Regression model is defined by the following equation (if it contains only one independent variable, then we have a univariate Logistic Regression model):

$$p(X_1, X_2, \ldots, X_n) = \frac{e^{(C_0 + C_1 \bullet X_1 + \ldots + C_n \bullet X_n)}}{1 + e^{(C_0 + C_1 \bullet X_1 + \ldots + C_n \bullet X_n)}} ,$$

where $\pi(X_1, X_2, \ldots, X_n)$ is the probability that Y = 1 (therefore $1-\pi(X_1, X_2, \ldots, X_n)$ is the probability that Y = 0). Coefficients C_i are estimated through Maximum Likelihood estimation. We use the following two statistics to describe the experimental results:

- *p, the statistical significance of the logistic regression coefficients*: the level of significance of the Logistic Regression coefficient C_i provides the probability that C_i is different from zero by chance, i.e., X_i has an impact on π.
- *R^2, the goodness of fit*, not to be confused with least-square regression R^2—they are built upon very different formulae, even though they both range between 0 and 1 and are similar from an intuitive perspective. The higher R^2, the higher the effect of the model's independent variables, the more accurate the model. However, as opposed to the R^2 of least-square regression, a high value for R^2 is rare for logistic regression. R^2 may be described as a measure of the *proportion of total uncertainty* that is attributed to the model fit.

Case Study 2
The effort to write a class was measured as the person-days needed to write and modify the class. The amount of change of a class was measured as the number of versions of the class.

6.5 Validate Dependent Measures

In Section 5.5, the validation of independent measures was shown to consist mainly of checking the measures against a set of properties. , This may not always be the case for dependent measures, since they usually cannot be precisely characterized by a set of specific mathematical properties. To provide support for the validity of the measures, it is more useful to look carefully at the data collection procedures and their integration into the development process. Typical examples are: checking defect counting procedure to assess their completeness and consistency, assessing whether an effort measurement includes all relevant activities or is broken down into activities that are clearly identifiable in the development process. Such a validation is qualitative in nature and requires some detailed knowledge of the development process in place. However, in some cases it is still possible to use theoretical validation techniques. For instance, it can be shown that the probabilities provided by Logistic Regression satisfy the properties for probabilities of Section 6.1.2.

7. Hypothesis Refinement and Verification

In this high-level step, the original empirical hypotheses (see Section 4.2) must be instantiated and possibly refined, based on the specific measures devised for both independent and dependent attributes (Section 7.1). Then, data must be collected to verify these hypotheses.

7.1 Instantiate and Refine Empirical Hypotheses

Description

The empirical hypotheses set in the *Define Empirical Hypotheses* step involve attributes. They need to be instantiated with the measures defined for both the independent and the dependent attributes. Once measures have been defined, it is possible to state more precise empirical hypotheses on the kind of relationships between independent and dependent measures, consistent with the initial empirical hypotheses. For instance, instead of stating a generic non-decreasing monotonic correspondence between the measure of the independent attribute and the measure of the dependent attribute, one can hypothesize a specific functional form, e.g., non-decreasing linear. This refinement need not be carried out right after the instantiation of the empirical hypotheses, but can be delayed until further evidence is available on the verification of the instantiated empirical hypotheses.

Examples

Case Study 1
Instantiation of the empirical hypotheses means that the coupling measures are used as the independent variables in the Logistic Regression equation—the dependent variable being a measure of fault-proneness. No further refinement of the empirical hypotheses was carried out.

Case Study 2
The empirical hypotheses were instantiated by replacing the axiom size attribute appearing in the empirical hypothesis with the corresponding measure. After the instantiated empirical hypothesis was confirmed by the empirical validation (see Section 7.2), we proceeded to build a refined empirical hypothesis, by hypothesizing a linear correlation between the independent and the dependent variables. The verification of this hypothesis is also shown in the next step (Section 7.2).

Discussion

The refined hypotheses should logically imply the original instantiated hypotheses, otherwise there is a contradiction. For instance, suppose that the instantiated hypothesis establishes a non-decreasing, monotonic correspondence between the measure of the independent attribute and the measure of the dependent attribute. If this hypothesis is supported by the empirical verification, one should not refine it by hypothesizing that there is a decreasingly linear correspondence between the measure of the independent attribute and the measure of the dependent attribute. On the other hand, if the instantiated hypothesis is not supported by the empirical verification, one should re-examine the actions carried out during the whole measure definition process, to find out where the problem lies.

Sometimes, the functional correspondence law between independent and dependent measures is already provided in the initial empirical hypotheses. However, this is possible only when the measures that will be used to quantify attributes are already known, so it is not necessary to go through all the steps of our measure definition approach, since the measures are already given.

7.2 Verify Empirical Hypotheses

Description

Based on the measures defined, data collection must be carefully designed and carried out so as to make sure that the data collected are actually consistent with the defined measures and that all the necessary information is collected to carry out the empirical validation.

Then, the data collected must be used to validate the refined empirical hypotheses upon which the measures are defined. The procedure to follow for experimental validation significantly depends on the purpose of measurement. With a prediction focus, one needs to validate a stated statistical relationship between independent and dependent measures. If the refined empirical hypotheses are not supported by the experimental results, one needs to reconsider all the steps and identify the causes of the problem, as we outline in Section 8.

Examples

We address data collection for each case in the corresponding part. As for the validation of the empirical hypotheses in both cases, we first carried out outlier analysis to remove single data points that could overly influence the results, so that they do not depend on one or a few data points. We used *univariate* logistic regression to evaluate the impact of each measure in isolation on the dependent variables. Only variables whose statistical significance is less than 5% (i.e., there is at least a 95% probability that they actually have an impact on the dependent variable) and explain a significant percentage of uncertainty in the data set have been considered as relevant predictors. Then, to build more accurate classification models, we performed *multivariate* logistic regression, to evaluate the relative impact of those measures that had been assessed sufficiently significant in the univariate analysis and also explained a significant percentage of the variation in the data set (i.e., sufficiently high values for R^2).

In Case Study 2, we also performed ordinary least-squares (OLS) linear regression for both specification effort and change analysis, to verify the refined empirical hypotheses.

Case Study 1
Data collection was carried out by using the software defect collection and tracking mechanism in place at GSFC, which had been used for several years before our empirical study was carried out. Thus, we were confident that the problem reports were consistent with each other. We took into account the defects generated during integration testing, i.e., those that were subject to reporting. The problem reports also identified the software parts of the fault that caused the problem. We developed a tool to extract the product measures on the high-level designs.

The available data allowed us to validate the instantiated empirical hypotheses and we now summarize the results we obtained on the three projects we studied (more details are in [19]).

In univariate analysis, DIC and TIC were found to be significant predictors (at the 5% significance level) for fault-proneness of software parts across the three projects, except for TONS, in which DIC and TIC were not significant. A discussion on the reasons for this is provided in Section 8. Also, the signs of the coefficients C_1 estimated for the univariate Logistic Regression equations confirmed the instantiated empirical hypotheses, i.e., the fault-proneness of a software part increases when DIC and TIC increase. The proportion of explained variation (R^2) did not exceed 0.15, which is still acceptable in the context of Logistic Regression.

Multivariate analysis showed that DIC and TIC are part of a multivariate Logistic Regression model, along with two other variables, the number of software parts imported (USES relation) by a software part, and the average depth of the hierarchy of units contained in a software part. These two additional measures too were defined via the GQM/MEDEA approach. For multivariate analysis, R^2 was 183%, 171%, and 269% of the best univariate R^2, with a maximum value of 0.43 for TONS.

Case Study 2
Data collection was carried out by recording the different versions and effort through a simple reporting mechanism and by computing the specification measures by hand, since the specification's size was not too large.

To use Logistic Regression, we discretized the effort and version ranges (the dependent variables in the Logistic Regression equation), with the first quartile, the median, and the third quartile as thresholds.

Univariate Logistic Regression showed that Axioms was a significant predictor for both effort and versions, along with other measures that were also defined via the GQM/MEDEA approach. At any rate, Axioms was by far the most influential variable: its R^2 value was more than 0.30. The sign of the C_1 coefficients estimated for the univariate Logistic Regression equations also confirmed the instantiated empirical hypotheses. We also built multivariate Logistic Regression models for predicting the version quartile of a TRIO+ class, according to the discretization of the version range. This first analysis allowed us to empirically validate the instantiated unrefined empirical hypotheses.

We then carried out an OLS linear regression for both effort and changes. The statistical analysis confirmed the results obtained with Logistic Regression. In particular, we were able to build a multivariate OLS model for the number of versions whose linear regression R^2 was 0.834.

Discussion

Our examples involve only two techniques (Logistic Regression and OLS Linear Regression), but a number of analysis techniques, both univariate and multivariate [12, 38, 11, 9, 23], exist in the statistical and machine learning literature. These techniques aim at identifying and modeling relationships between the independent measures and dependent measures. They differ in the model structure they assume, in their data fitting algorithms, and their ease of interpretation. Different types of data and application purposes may warrant the use of different modeling techniques.

The instantiated and possibly refined empirical hypotheses are validated through statistical test of hypotheses. This statistical test of hypotheses may involve quantities such as coefficients that are a part of regression models, for which standard tests are usually used.

Since there is extensive literature on the empirical validation of software measures subject, we will not discuss this step any further. For instance, the interested reader may consult [12, 38, 14] for a detailed discussions of issues related to the empirical validation of software measures.

8. Identifying Problems

One of the strengths of a well-defined measure definition approach is the support it can provide in locating and identifying the causes of problems. Based on our application cases, we now discuss the most common issues encountered and provide a few examples in which GQM/MEDEA helped us identify problems and the corresponding steps by which they were introduced.

From a general perspective, GQM/MEDEA is helpful mainly because it makes explicit all the decisions involved in planning a measurement activity that aims at building a prediction model. This facilitates the evaluation and refinement of these measurement activities. To illustrate the point, this is similar to the design stage of software development itself where design artifacts record design decisions, which can then be retrieved, reviewed, and changed.

Steps: Define and Verify Empirical Hypotheses
Other hypotheses were formulated and verified, in addition to those shown in this paper. For instance, we stated the following empirical hypotheses for Case Studies 1 and 2, respectively.

1. The larger the size of the software part’s high-level design, the higher its fault proneness.
2. The larger the coupling of a TRIO+ class, the higher the amount of changes on that class.

These hypotheses seemed to be well-founded, since there is extensive literature that supports the idea that size is related to faults (Case Study 1) and that a system should be made of loosely coupled modules (Case Study 2). However, applying GQM/MEDEA, we were not able to find any empirical evidence to support the instantiated hypotheses.

It is not a surprise that some hypotheses will not be confirmed. However, one cannot reject them without first considering the following issues, which are commonly encountered in practice:

1. The relationship hypothesized may be true but the measures may have poor construct validity, i.e., they may not measure accurately the concept they purport to measure. Because relatively little experience exists as compared to other, more mature empirical fields, construct validity is an important problem in software measurement.
2. The available sample or sample variance may be too small to detect the effect at a statistically significant level, i.e., there is insufficient statistical power [14].
3. The functional form assumed for the relationship may be incorrect, e.g., exponential instead of linear.
4. Factors, other than the independent attribute considered, may blur the relationship which would only be visible by looking simultaneously at several relationships, i.e., by multivariate analysis.

There are several explanations for the negative results we obtained for either hypothesis. In Case Study 1, we believe that none of the four issues above are plausible, so the size of the high-level design of a software part was not a relevant factor in the prediction of the software part's fault-proneness in the final product. (1) The measure we used for high-level design size was straightforward and satisfied all properties for size. (2) The sample size and variance were large enough so that they cannot be claimed to justify the negative result. (3) Logistic Regression does not constrain the relationship between the independent measures and the dependent measure much. (4) Multivariate analysis did not support the idea that size could be related to fault-proneness. In general, before concluding that a hypothesis is not supported by the available data, it is important that all plausible causes should be considered.

The most plausible explanation for Case Study 2 is related to issue (2), i.e., the sample size was relatively small and only strong (at the 5% significance level) relationships could be detected with the power of the statistical tests we used. Thus, we cannot exclude that the hypothesis holds, even though the relationship was not significant even at the 10% level. Further studies should be performed to ascertain whether this relationship is present.

In Case Study 1 the lack of significance of TIC in the system TONS (at the 5% level) can be explained by a lack of variability. TIC's standard deviation as compared to its mean is much smaller in TONS than in the other two systems studied in [19]. TIC’s mean and median were also smaller and so was the proportion of faulty software parts. This seems to confirm that controlling TIC would indeed reduce defects. However, no explanation was found for the lack of significance of DIC in TONS. This shows that some problems may remain unsolved.

In general, it is not easy to make sure that lack of statistical power is the cause of a negative result. However, when there is evidence that this is the case, it is usually not possible to increase the number of data points in a correlational study like the one we describe in this paper. Therefore, one should try to replicate the empirical study, although caution should be used to make sure that differences across environments or software organizations do not affect the results of the replications.

The somewhat mediocre values for R^2 in Case Study 1 can be explained by the fact that we were assessing the influence of the high-level design phase on the final product's fault-proneness. High values for R^2 in Case Study 1 would imply that one can accurately predict the final product's fault-proneness based only on high-level design information. This would imply that the coding phase has only a marginal influence on the final product's fault-proneness. Since this is unlikely to be true, we should not expect a very high value of R^2. In any case, we have identified a number of high-level design characteristics that are correlated with the final product's fault-proneness in a statistically significant way. In Case Study 2, we studied the relationships between characteristics of specifications and the fault-proneness and effort of the specification phase. Therefore, we could expect to identify more accurate models, as we actually did.

Step: Formalize Independent Attributes
In Case Study 1, the properties for complexity measures of [42] did not satisfy *our intuition* about the attributes contained in the empirical hypotheses (e.g., coupling). Therefore, we formalized our own intuition through a different set of properties for each relevant attribute [18]. We "reused" these sets of properties in Case Study 2.

From a general perspective, independent attributes should be defined in a way that satisfies any specific intuition we may have about them, at least until a well accepted standard emerges. Well-defined independent attributes help ensure the construct validity of their measures and avoid difficulties in the validation of the empirical hypotheses. We believe that a property-based approach provides an adequate and rigorous characterization of independent attributes [18], especially internal product attributes. However, given the current state of the art, it is often the case that existing properties do not fit one’s own specific intuition and must be refined in the particular context where the study is performed.

Step: Identify Abstractions (for defining independent measures)
In Case Study 1, we started by using the interaction graph (see Figure 6(b)) for cohesion (the other independent attribute we studied). We did not find it suitable for cohesion, because it did not adequately capture the kind of relationships we wanted to capture among data declarations and functions. Therefore, we introduced cohesive interactions and cohesive interaction graphs [19]. The need for this abstraction became apparent during the *Instantiate and Refine Properties* step.

In general, when attribute properties cannot be properly expressed with the defined abstractions, it is important to modify the existing abstractions to allow for the definition of properties that are as specific as intuition permits.

Step: Define Independent Measures
Before defining the RCI measure for cohesion in Case Study 1 [19], we introduced a few other measures that turned out not to comply with the properties we believe necessary for cohesion measures. Therefore, we discarded them. This problem arose during the *Verify Independent Measures* step, well before the *Verify Empirical Hypotheses* step. Our objective, once we identified cohesion as a relevant attribute, was to verify whether cohesion was really related to fault-proneness. Accepting measures that did not satisfy our intuition on cohesion would have made the interpretation of results difficult, if not incorrect.

In general, one important lesson learned is that it is easy, even for experienced researchers and practitioners, to be misled and define measures that do not show appropriate properties. Measures must be systematically analyzed and verified to ensure they are measuring the attributes right. Sets of properties, as defined in [18], are designed to facilitate such a verification procedure.

In this section, we have illustrated some of the lessons learned and advantages of using a systematic approach to defining, assessing, and refining measures aimed at building prediction systems. Too often in our field, practitioners and researchers rush through the measure definition steps to focus on empirical prediction results. This is a risky approach that may lead to results that are unexpectedly negative, positive but spurious, difficult to interpret, or difficult to build on in subsequent studies.

9. Conclusions and Future Work

Software measures need to be defined in a rigorous and disciplined manner based on a precisely stated experimental goal, assumptions, properties, and thorough experimental validation. We propose a practical approach to define software measures that implements these principles (GQM/MEDEA) and integrates many of the contributions from the literature. It results from experience and has been validated through realistic examples and field studies [17, 9, 12, 20, 19, 21]. Thus, this paper completes our previous research [18, 19] and describes, in an operational and practical way, some of the lessons learned resulting from our experience.

As for its specific contributions, our framework

- provides a detailed description of the various activities involved in the definition of measures and of the information flow among these activities;
- links the definition of measures to corporate goals and the development environment;
- shows that measures should not be defined *per se*, but they should be defined in the context of a theory;
- helps better justify, interpret, and reuse measures;
- helps identify problems that may arise during the definition of measures, taking in consideration that it is a highly human-intensive process;
- provides a conceptual model that can be used for implementing the schema of a repository that contains all the knowledge relevant to measurement.

Our future work encompasses a more detailed study and validation of each of the steps involved in the measure definition approach. Specifically, we need to

- better understand how experimental results can be used to guide the refinement of measures based upon our increased understanding of the development processes and their evolution;
- better identify what can be reused across environments and projects, e.g., measures, assumptions, measurement concepts, abstractions;
- provide more accurate guidelines to help experimenters make some of the subjective decisions involved in the GQM/MEDEA approach.

Acknowledgments

We would like to thank William Agresti, Dieter Rombach, Yong-Mi Kim, Bryan Hsueh, Wolfgang Heuser, Oliver Laitenberger, and Manoel Mendonça for their help in reviewing the early drafts of this paper. This work was supported in part by NASA grant NCC5170, NSF grant CCR9706157, NSF Grant CCR9706151, NSERC Canada, CNR, and MIUR.

References

All ISERN technical reports can be found on: http://www.iese.fhg.de/ISERN.

[1] V. Barnett and T. Price, *Outliers in Statistical Data*, John Wiley & Sons, 3rd edition, 1995.

[2] R. Van Solingen "The Goal/Question/Metric Approach," Encyclopedia of Software Engineering - 2 Volume Set, pp. 578-583, Copyright by John Wiley & Sons, Inc., 2002.

[3] V. R. Basili, "The Experience Factory and its Relationship to Other Improvement Paradigms," 4th European Software Engineering Conference (ESEC) in Garmish-Partenkirchen, Germany. The Proceedings appeared as the Springer-Verlag Lecture Notes in Computer Sciences Series 717, September 1993.

[4] V. R. Basili, G. Caldiera, D. H. Rombach, "The Experience Factory," Encyclopedia of Software Engineering - 2 Volume Set, pp. 511-519, Copyright by John Wiley & Sons, Inc., 2002.

[5] V. R. Basili and D. H. Rombach, "The Tame Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, 14 (6), pp. 758-773, June 1988.

[6] V. R. Basili and D. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Eng.*, 10 (11), pp. 758-773, November 1984.

[7] V. R. Basili, M. Zelkowitz, F. McGarry, J. Page, S. Waligora and R. Pajerski, "Special Report: SEL's Software Process-Improvement Program," *IEEE Software*, Volume 12, Number 6, pp 83-87, November 1995.

[8] J. Bieman, A. Baker, P. Clites, D. Gustafson, and A. Melton, "A Standard Representation of Imperative Language Programs for Data Collection and Software Measures Specification," *Journal of Systems and Software*, vol. 8, pp. 13-37, 1988.

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

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

[11] L. Briand, V. R. Basili, W. Thomas, "A Pattern Recognition Approach for Software Engineering Data Analysis," *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

[12] L. Briand, J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems", forthcoming in *Advances in Computers*, Academic Press, 2002.

[13] L. Briand, C. Differding, D. Rombach, "Practical guidelines for measurement-based process improvement," *Software Process Improvement and Practice*, 1997. Also available as Technical Report ISERN-96-05, Fraunhofer Institute for Experimental Software Engineering, Germany, 1996.

[14] L. Briand, K. El Emam, S. Morasca, "Theoretical and Empirical Validation of Software Product Measures," Technical Report ISERN-95-03, Fraunhofer Institute for Experimental Software Engineering, Germany, 1995.

[15] L. Briand, K. El Emam, S. Morasca, "On the Application of Measurement Theory to Software Engineering," *Empirical Software Engineering - An International Journal*, Vol. 1, No 1, 1996.

[16] L. Briand, S. Morasca, "Software Measurement and Formal Methods: A Case Study Centered on TRIO+ Specifications," *in Proc. ICFEM'97*, November 12-14, 1997, Hiroshima, Japan.

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

[18] L. Briand, S. Morasca, V. R. Basili, "Property-based Software Engineering Measurement," *IEEE Trans. Software Eng.*, 22 (1), January, 1996.

[19] L. Briand, S. Morasca, V. R. Basili, "Defining and Validating Measures for Object-based High-Level Design," *IEEE Trans. Software Eng.*, 25 (5), pp. 722 – 741, September/October 1999.

[20] L. Briand, J. Wüst, S. Ikonomovski, H. Lounis, "A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: An Industrial Case Study," *proceedings of ICSE'99*, Los Angeles, USA. A full version is also available as a technical report ISERN-98-29.

[21] L. Briand, J. Wüst, John W. Daly, V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems", *Journal of Systems and Software*, 51(2000) p 245-273.

[22] T. De Marco, *Structured Analysis and System Specification*, Yourdon Press Computing Series, 1978.

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

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

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

[26] N. Fenton and S. Pfleeger, Software Metrics, *A Practical and Rigorous Approach*, Thomson Computer Press, 1996.

[27] D. A. Gustafson et al., "Software measure specification," Proc, SIGSOFT '93, *SIGSOFT Software Engineering Notes*, vol. 18, no. 5, pp. 163-168.

[28] P. G. Hoel, *Introduction to Mathematical Statistics*, Wiley and Sons, 1984.

[29] D. Ince, M. Shepperd, "System Design Metrics: a Review and Perspective *Proc.12th Int. Conf. on Software Engineering*, pages 23-27, 1988

[30] ISO/IEC 9126-1, "Software Product Quality," Part 1: Quality Model.

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

[32] M. Lewis-Beck, Applied Regression, SAGE publications, 1980

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

[34] S. Morasca, L. Briand, "Towards a Theoretical Framework for Measuring Software Attributes," *in Proc. IEEE Symposium on Software Metrics*, Albuquerque, 1997

[35] A. Morzenti and P. San Pietro, "Object-Oriented Logical Specification of Time-Critical Systems," *ACM Transactions on Software Engineering and Methodology*, Vol.3, n.1, pp. 56-98, January 1994.

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

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

[38] N. F. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410-422, May 1992.

[39] P. Spector, *Research Designs*, SAGE publications, 1981.

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

[41] Specification of the Unified Modeling Language (UML), version 1.4, http://www.omg.org/

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

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

View publication stats