Defining and Validating Measures for Object-Based High-Level Design

Lionel C. Briand, Sandro Morasca, *Member*, *IEEE Computer Society*, and Victor R. Basili, *Fellow*, *IEEE*

Abstract—The availability of significant measures in the early phases of the software development life-cycle allows for better management of the later phases, and more effective quality assessment when quality can be more easily affected by preventive or corrective actions. In this paper, we introduce and compare various high-level design measures for object-based software systems. The measures are derived based on an experimental goal, identifying fault-prone software parts, and several experimental hypotheses arising from the development of Ada systems for Flight Dynamics Software at the NASA Goddard Space Flight Center (NASA/GSFC). Specifically, we define a set of measures for cohesion and coupling, which satisfy a previously published set of mathematical properties that are necessary for any such measures to be valid. We then investigate the measures' relationship to fault-proneness on three large scale projects, to provide empirical support for their practical significance and usefulness.

Index Terms—Measurement, object-based design, high-level design, Ada, cohesion, coupling.

1 Introduction

Software measures can help address the most critical issues in software development and provide support for planning, predicting, monitoring, controlling, and evaluating the quality of both software products and processes [15], [23]. Most existing software measures attempt to capture attributes of the software code [23]; however, software code is just *one* of the artifacts produced during software development, and, moreover, it is only available at a late stage. It is widely recognized that the production of better software requires the improvement of the early development phases and the artifacts they produce. The production of better specifications and designs reduces the need for extensive review, modification, and rewriting not only of code, but of specifications and designs. As a result, a software organization can save time, cut production costs, and raise the final product's quality.

Early availability of measures is a key factor in the successful management of software development, since it allows for:

1) the early detection of problems in the artifacts produced in the initial phases of the life-cycle (specification and design documents) and, therefore, reduction of the cost of change—late identification and correction of problems are much more costly than early ones;
2) better software quality monitoring from the early phases of the life-cycle;
3) quantitative comparison of techniques and empirical refinement of the processes to which they are applied;
4) more accurate planning of resource allocation, based upon the predicted quality of the system and its constituent parts.

In this paper, we focus on measures for the high-level design of object-based[1] software systems, to study whether information available at this development stage can be used to support the issues raised in points 1), 2), 3), and 4). We worked in the context of high-level designs for Flight Dynamics software, written in Ada83 [22], in the Software Engineering Laboratory at NASA Goddard Space Flight Center (GSFC). Our goal was to

> ***define and validate a set of high-level design measures to evaluate the quality of the high-level design of a software system with respect to its fault-proneness and understand which high-level design attributes are likely to make software fault-prone.***

We set a number of experimental hypotheses that were believed to be true in the environment of our study. In our study, we define three families of measures to set the hypotheses in measurable terms. These hypotheses were empirically validated based on three projects conducted at the NASA/GSFC. As with many empirical studies, some of the hypotheses were supported by the empirical results, while others were not. In this paper, due to space constraints, we only report those hypotheses and measures that were supported by the empirical results.

Specifically, we introduce and theoretically validate, based on the properties of [12], a family of measures for cohesion and coupling of high-level object-based software designs.

- *L.C. Briand is with the Fraunhofer-Institute for Experimental Software Engineering, Technologiepark II, Sauerwiesen 6, D-67661 Kaiserslautern, Germany. E-mail: briand@iese.fhg.de.*
- *S. Morasca is with the Dip. di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy. E-mail: morasca@elet.polimi.it.*
- *V.R. Basili is with the Computer Science Department, University of Maryland, College Park, MD 20742. E-mail: basili@cs.umd.edu.*

Manuscript received 9 June 1994; revised 17 Feb. 1998.
Recommended for acceptance by not listed.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 101168.

1. Object-based systems differ from object-oriented systems in that inheritance is not allowed.

0098-5589/99/$10.00 © 1999 IEEE

Our measures focus and are based on one specific facet of cohesion and coupling, i.e., that related to declaration links among data and subroutines appearing in high-level design module interfaces. Therefore, our measures are not meant to capture all aspects of cohesion and coupling. For the sake of comparison and completeness, we also define two simpler measures based on USES and IS_COMPONENT_OF [25] relationships between modules. This appears necessary at this stage of knowledge, where we can only rely on very limited theoretical and empirical grounds to help us identify interesting concepts, relationships and objects of study. If our measures add complexity to the analysis, they should also be complementary to simpler design measures already proposed in the literature. One of the results of this investigation is to provide directions for focusing our research on a smaller set of strategies and concepts.

A number of studies have been published on software design measures in recent years. It has been shown that system architecture has an impact on maintainability and fault-proneness [26], [24], [38], [30], [39], [16], [40], [41], [43], [1], [17], [2], [44]. These studies have attempted to capture the design attributes affecting the ease of maintaining and debugging a software system. Most of the design measures are based on information flow between subroutines or declaration counts. We think that, even though they provide interesting insights into the program structure, these should not be the only strategies to be investigated, since many other types of program features and relationships are a priori worth studying. Moreover, there is a need for comparison among strategies in order to identify worthwhile research directions and build accurate quality prediction models.

In addition, the success and widespread diffusion of object-oriented software systems have drawn a good deal of interest towards the study of the attributes of object-oriented software systems. A number of studies have been published (see for instance [19], [8], [28] and [5], [6] for an extensive survey). These studies generally deal with the proposal of new measures or the reuse of existing ones in the framework of object-oriented software code. Our study goes one step in the direction of object-orientation, *at the high-level design stage*, in that it addresses *object-based* systems. Therefore, we take into account several important characteristics of object-oriented software, with one important exception—inheritance.

The paper is organized as follows. In Section 2, we concisely outline the overall structure of our study and explain the process we have carried out and its rationale. Section 3 contains the basic definitions and concepts that are used in the paper. The cohesion and coupling measures we introduce are presented in Sections 4 and 5, respectively. Based on the USES and IS_COMPONENT_OF relationships [25], we also define two simpler measures (Section 6), which are commonly proposed in the literature and against which we wish to compare our cohesion and coupling measures. These two measures were part of a larger set but turned out to be the only ones yielding positive results as indicators of fault-proneness (see [10] for further details). Empirical validation of the measures is shown in Section 7. In Section 8, we summarize the lessons we have learned, and outline directions for future research activities.

2 Outline of the Study

We now describe the measurement activities we carried out, to provide the reader with a better interpretation framework for our study. The steps we carried out follow the scientific method and concern the setting of experimental goals and hypotheses, the definition of appropriate measures, and the theoretical and experimental validation of those measures. The steps below were basically executed in a sequential fashion. However, some steps were, to some extent, executed in parallel, e.g., steps 3 and 4; in addition, the need occasionally arose in a few points of the execution to go back to steps that had been already executed.

1) ***Establish measurement goals***. Empirical software engineering fosters the improvement of software products and processes. In this context, measurement should be seen as a tool for acquiring information that can be useful for specific improvement purposes. Thus, precise measurement goals should be set, to ensure specific improvement issues of interest are addressed. It is our opinion that, at this stage, the definition of universal measures (like in physical sciences) is a long-term goal, which, however, is only achievable (if at all) after we gain better insights into specific environments and from specific perspectives in the short term. Therefore, the definition of a measure should be driven by both the characteristics of the context or family of contexts in which it is used and one or more clearly stated goals that it helps reach.

   The goal of our study was to analyze the high-level design of three software systems in order to understand which high-level design attributes are likely to make software fault-prone in our application context, NASA/GFSC.

2) ***Set experimental hypotheses***. Experimental hypotheses, derived from the measurement goals, are necessary to define measures that are somewhat supported by an underlying theory to be confirmed or disconfirmed. Thus, we avoid a random search for statistical significance. Experimental hypotheses establish a link between the attribute of interest (software code fault-proneness, in our case) and some attribute of the object of study, e.g., size, complexity, cohesion, coupling of software high-level design.

   Each measure we introduce in our study is accompanied by an experimental hypothesis. However, we do not claim that these hypotheses are universally true in any environment: a priori, they may not even be true in our environment, since they can be disconfirmed by the empirical validation. Also, other hypotheses could be set: other people may come up with different hypotheses in the same environment, since our hypotheses capture our beliefs. In addition, we do not assume that all of these experimental hypotheses are equally important towards our experimental goal, i.e., not all of the attributes we take into account have an equal impact on software fault-proneness. In this paper, we will only report on those hypotheses that were confirmed by the empirical validation (Section 7), and, therefore, we will only introduce those measures

that allowed us to quantify these hypotheses. The reader interested in the negative results of this study may consult [10].

3) ***Characterize formally the attributes to be studied***. Experimental hypotheses are stated in terms of attributes, which are to be quantified by means of measures. The introduction of appropriate measures is facilitated by the availability of precise definitions for the attributes of interest. Unfortunately, such attributes, e.g., size, complexity, cohesion, coupling, are hardly ever defined in a precise and unambiguous way, if they are defined at all. However, approaches have appeared in the recent literature to provide these attributes with less fuzzy and ambiguous definitions, using mathematical properties to characterize them [42], [12].

   In our study, we have used an instantiation of the property-based approach of [12] for our object-based Ada context, to provide theoretical support for the definition of our measures of cohesion and coupling based on data declaration dependency links. These properties allow us to characterize—to the best of our understanding and knowledge—the two attributes, and provided us with guidance for measure definition. They provide supporting evidence that the measures are theoretically valid, i.e., we measure what we purport to measure (see step 5).

   We want to point out that acceptance of our cohesion and coupling properties is, to some extent, a subjective matter, as with any other set of properties or rationalization of informal concepts. Also, our properties are to be interpreted as necessary, but not sufficient. This is the case even for the most consolidated and well-known ones, such as the properties for distance. As a consequence, measures might be built that satisfy our properties but cannot be taken as sensible cohesion or coupling measures. However, we believe that, by providing desirable properties for the measures of cohesion and coupling, we have better clarified our ideas about cohesion and coupling. The reader has much more solid grounds on which he or she can either accept our ideas about cohesion and coupling, or reject them and replace them with other properties.

4) ***Identify abstractions of the object of study***. Appropriate representations (*abstractions* according to [37]) of the object of study are used in measurement to capture the information needed to build measures for the software attributes mentioned in the experimental hypotheses. Some examples of product abstractions are data flow graphs and control flow graphs. In our study, we use graphs based on dependency links between data and subroutines in high-level software design.

5) ***Define measures***. A measure is defined for capturing some intuitive concept [31], e.g., size, complexity, cohesion, coupling, such as those used in the experimental hypotheses. In our study, we define measures for cohesion and coupling based on dependency links among data and subroutines in high-level software design.

   The definition of sound measures requires that they be theoretically validated, to show that they actually quantify the attributes they purport to measure. This is argued for our cohesion and coupling measures because they satisfy the properties for those measures we established in step 3, and because they do not satisfy any set of properties for other attributes such as complexity or size. One of the goals of [12] was to define sets of properties to identify similarities and differences across software attributes.

   At any rate, as explained in step 3, some caution must be used in interpreting the results of our theoretical validation, as with any theoretical validation, due to the inherent degree of subjectivity in the formalization of intuition and the fact that properties are necessary but not sufficient. Therefore, the satisfaction of our cohesion and coupling properties cannot be strictly taken as conclusive evidence that the measures we define are cohesion and coupling measures, but only as supporting evidence. In addition, we do not claim nor believe that our measures are the "definitive" measures for cohesion and coupling. They address only one possible aspect of cohesion and coupling, and, even in our context, they will need further refinements.

6) ***Validate measures empirically***. The empirical validation of a measure actually entails the validation of the experimental hypotheses involving the attribute quantified by the measure. Empirical validation ascertains the practical usefulness of a measure in the studied environment, by showing if the attributes it measures, e.g., cohesion, influences an external quality attribute [23] of practical interest, e.g., fault-proneness, and the extent of this influence.

   In our empirical validation, based on data collected at the NASA/GSFC, we have applied a statistical technique to study the influence of cohesion and coupling on fault-proneness. Validation was facilitated by the fact that we had defined experimental goals and hypotheses at the beginning of the study. At any rate, the external validity of the experimental hypotheses and measures remains to be investigated in order to determine whether they are applicable to different environments and problem domains.

More details about the approach we have followed can be found in [11].

3 Basic Definitions

In this section, we first introduce the basic concepts and the terminology that we will use in the paper (Section 3.1). We then define interactions, the data dependency links on which our cohesion and coupling measures are based (Section 3.2).

3.1 Modules and High-Level Design

Our object of study is the high-level design of a software system. To define it, we will start from its elementary components: software modules. In the literature, there are two commonly accepted definitions of modules. The first one sees a module as a subroutine, and has been used in most of

the design measurement publications [35], [20], [26], [38], [40]. The second definition, which takes an object-oriented perspective, sees a module as a collection of type, data, and subroutine definitions, i.e., a provider of computational services [13], [25]. In this view, a module is the implementation of an Abstract Data Type (ADT), e.g., a package in Ada, a class in C++. In this paper, unless otherwise specified, we will use the term subroutine for the first category, and reserve the term module for the second category. Modules are composed of two parts: interface and body (which may be empty). The interface contains the computational resources that the module makes visible for use to other modules. The body contains the implementation details that are not to be exported.

Modules and subroutines may be related to each other by IS_COMPONENT_OF and USES relationships [25]. In general, module/subroutine *A* is related to module/subroutine *B* by an IS_COMPONENT_OF relationship if *A* is defined within *B*. Module/subroutine *A* is related to module/subroutine *B* by a USES relationship if *A* uses computational services that *B* makes available.

Modules and subroutines can be seen as the components of higher level aggregations, as defined below.

DEFINITION 1 (Library Module Hierarchy (LMH)). *A library module hierarchy is a hierarchy where nodes are modules and subroutines, arcs between nodes are IS_COMPONENT_OF relationships, and there is exactly one top level node, which is a module.*

In the remainder of this paper, we will define concepts and measures that can be applied to both modules and LMHs, which are the most significant syntactic aggregation levels below the subsystem level. For short, we will use the term *software part* (*sp*) to denote either a module or a LMH.

In the high-level design phase of a software system in our context, only module and subroutine interfaces and their relationships are defined—detailed design of module bodies and subroutines is carried out at low-level design time. Therefore, we define the high-level design of a software system as follows.

DEFINITION 2 (High-level Design). *The high-level design of a software system is a collection of module and subroutine interfaces related to each other by means of USES and IS_COMPONENT_OF relationships. Precise and formalized information on module or subroutine bodies is not yet available at this stage.*

3.2 Interactions

In this section, we will specifically focus on the dependencies among data declarations and subroutines, which can propagate inconsistencies when changes are made to a software system. In this context, data declarations may be types, variables, or constants. Those dependencies will be called interactions and will be used to define measures capturing cohesion and coupling within and between software parts, respectively.

There are four possible kinds of interactionsfrom:

1) data declarations to data declarations,
2) data declarations to subroutines,
3) subroutines to subroutines, and
4) subroutines to data declarations.

However, not all of these dependencies can be detected at high-level design time. Therefore, we will investigate the interactions from data declarations to data declarations or from data declarations to subroutines, which we may detect from the high-level design of a software system.

DEFINITION 3 (Data Declaration-Data Declaration (DD). Interaction). *A data declaration A DD-interacts with another data declaration B if a change in A's declaration or use may cause the need for a change in B's declaration or use.*

The DD-interaction relationship is transitive. If *A* DD-interacts with *B*, and *B* DD-interacts with *C*, then a change in *A* may cause a change in *C*, i.e., *A* DD-interacts with *C*. Data declarations can DD-interact with each other regardless of their location in the designed system. Therefore, the DD-interaction relationship can link data declarations belonging to the same software part or different software parts.

The DD-interaction relationships can be defined in terms of the basic relationships between data declarations allowed by the language, which represent direct DD-interactions, i.e., not obtained by virtue of the transitivity of interaction relationships. Data declaration *A* directly DD-interacts with data declaration *B* if *A* is used in *B*'s declaration or in a statement where *B* is assigned a value. As a consequence, as bodies are not available at high-level design time in our application context, we will only consider interactions detectable from the interfaces.

DD-interactions provide a means to represent the dependency relationships between individual data declarations. Yet, DD-interactions per se are not able to capture the relationships between individual data declarations and subroutines.

DEFINITION 4 (Data Declaration-Subroutine (DS) Interaction). *A data declaration DS-interacts with a subroutine if it DD-interacts with at least one of its data declarations.*

Whenever a data declaration DD-interacts with *at least one* of the data declarations contained in a subroutine interface, the DS-interaction relationship between the data declaration and the subroutine can be detected by examining the high-level design. For instance, from the Ada-like code fragment in Fig. 1, it is apparent that both type *T1* and object *OBJECT11* DS-interact with procedure *SR11*, since they both DD-interact with parameter *PAR11*, which belongs to procedure *SR11*'s interface data declaration.

For graphical convenience, both sets of interaction relationships will be represented by directed graphs, the *DD-interaction graph*, and the *DS-interaction graph*, respectively. In both graphs (see Fig. 2, which shows DD- and DS-interaction graphs for the code fragment of Fig. 1), data declarations are represented by rounded nodes, subroutines by thick lined boxes, modules by thin lined boxes, and interactions by arcs.

```
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;
```

Fig. 1. Ada-like code fragment.

The notion of interaction can be applied to other object-based design methods and formalisms such as HOOD (one of the main object-based design methods [29]) with no basic changes. For instance, HOOD does not allow direct access to data in module interfaces, i.e., objects' provided interface. Using HOOD's terminology, data must be encapsulated in the internal part of each object (i.e., module) and must be accessed through public operations provided by the object. In that case, by looking at the visible part of a HOOD object description, we would analyze interactions between type definitions, constants, and operations, i.e., the same kind of information we have in our Ada context. When working with other design techniques, one can use all the available information on the interactions between the elements of a design. If mechanisms for describing such interactions exist, then one can apply our approach based on more information than is available in our case and in the HOOD case, and obtain more accurate models.

In this study, interactions are used to define measures for object-based high-level software design, which we introduce next. It is generally acknowledged that system architecture should have low coupling and high cohesion [20]. This is assumed to improve the capability of a system to be decomposed in highly independent and easy to understand pieces. However, the reader should bear in mind that high cohesion and low coupling may be conflicting goals, i.e., a trade-off between the two may exist. For instance, a software system can be made of small modules with a high degree of internal cohesion but very closely related to each other and, therefore, with a high level of coupling. Conversely, a software system

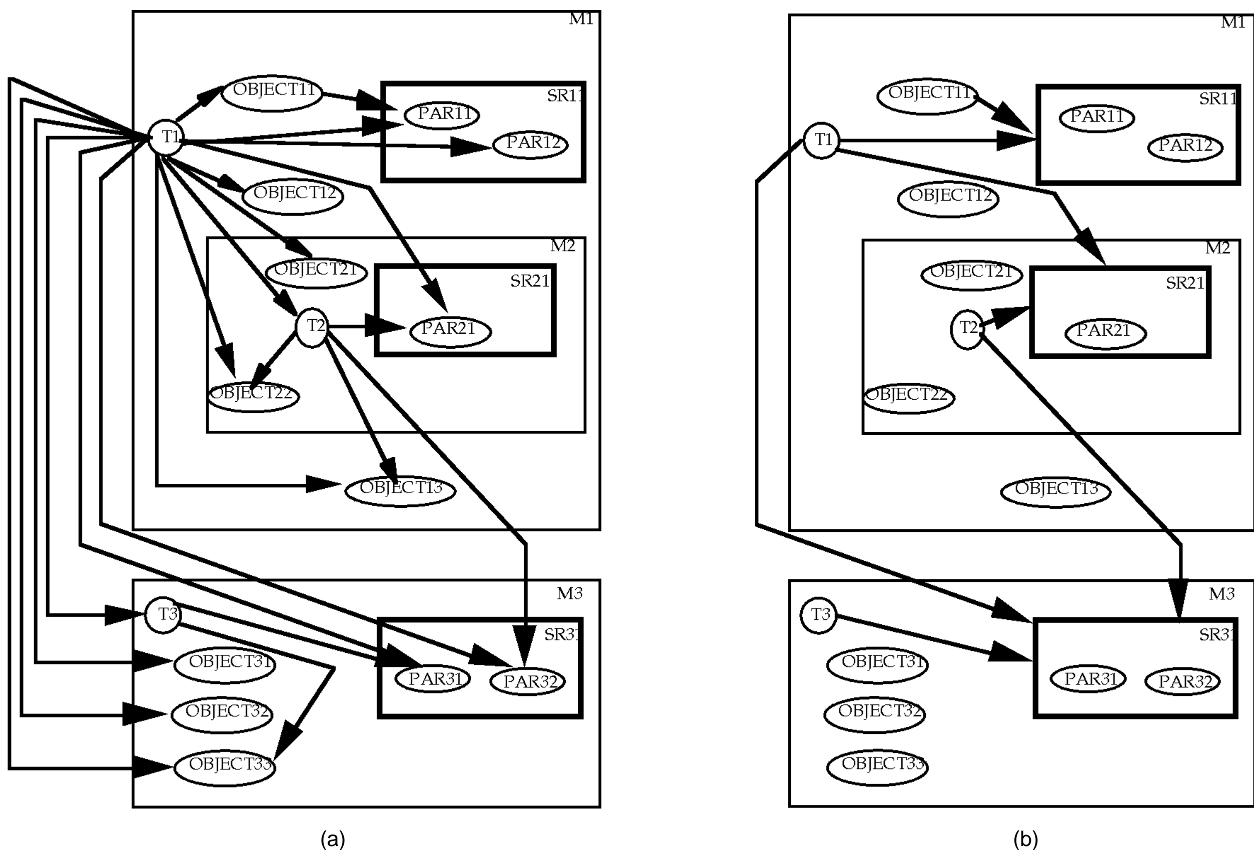


Fig. 2. Graphs for the code fragment in Fig. 1. (a) DD-interaction; (b) DS-interaction.

can be composed of few large modules, representing its subsystems, loosely related to one another, i.e., with low coupling, but with a low degree of internal cohesion as well. Moreover, high cohesion and low coupling are not the only factors to be taken into account when designing a software system. Other issues, e.g., reuse, must be taken into account as well.

4 Interaction-Based Measures for Cohesion

Consistent with the objectives stated above, cohesion is defined here as the degree to which data declarations and subroutines of a module are conceptually related, based on information known at the end of high-level design. In order for cohesion measurement to be usable on real-scale software systems, these conceptual relationships are to be approximated through syntactical relationships (i.e., interactions), which can be automatically detected through static analysis. Since we place ourselves at the end of high-level design in an object-based context, we focus on module-level cohesion and not on subroutine-level cohesion.

We introduce an experimental hypothesis (*H-CH*), which provides the motivations for defining cohesion measures in our object-based context with respect to our experimental goal (Section 4.1). Then, for illustration convenience only, we depart from the order in the step sequence described in Section 3: We first describe the abstraction used to define interaction-based cohesion measures (Section 4.2), and then (Section 4.3) we describe the properties for cohesion measures we proposed in [12] instantiated on this abstraction. An interaction-based cohesion measure is introduced in Section 4.4. In Section 4.5, we discuss how to use additional information that may be available at high-level design time. The relation of our work with previous works on cohesion measurement is discussed in Section 4.6.

4.1 Experimental Hypothesis

In our application environment, cohesion measurement is motivated by the following experimental hypothesis.

Hypothesis H-CH*. A high degree of cohesion is desirable because information related to declaration and subroutine dependencies should not be scattered among irrelevant information. Data declarations and subroutines which are not related to each other should be encapsulated, to the extent possible, into different modules. As a result of such a strategy, we expect the software parts to be less fault-prone.*

This hypothesis establishes a link between two software attributes: cohesion and fault-proneness. Its empirical validation requires that we introduce measures to capture cohesion and fault-proneness quantitatively. Fault-proneness will be quantified as the likelihood of a module to be faulty. Since we believe that this definition of a measure for fault-proneness is much more immediate and readily acceptable than the definition of a measure for cohesion or coupling, we now show how we introduced cohesion measures in the context of our study.

4.2 Abstraction Definition

Consistent with the definition of Abstract Data Type/Object, data declarations and subroutines should show some kind of interaction between them, if they are conceptually related. Therefore, we are interested in evaluating the tightness of the interactions between data declarations and data declarations or data declarations and subroutines declared in a module interface. We will capture this by means of *cohesive* interactions and the graph that they give rise to, the *cohesion interaction graph.*

Definition 5 (Cohesive Interactions). *The set of cohesive interactions in a module* m*, denoted by* CI(m)*, is the union of the sets of DS-interactions and DD-interactions involving exclusively data declarations and subroutines within m, with the exception of those DD-interactions between a data declaration and a subroutine formal parameter.* M(m) *will denote the maximal set of cohesive interactions in module* m*. It is obtained by linking every data declaration of module* m *to every other data declaration and subroutine of* m *with which it can interact.*

Definition 6 (Cohesive Interaction Graph). *Given a module* m*, the* Cohesive Interaction Graph *is the directed graph whose set of nodes is composed of the data declarations and the subroutines declared in module* m's *interface and whose set of arcs is the set* CI(m) *of module* m's *cohesive interactions.*

We use the Cohesive Interaction Graph as the abstraction on which we define our cohesion measures.

We do not consider the DD-interactions linking a data declaration to a subroutine parameter as relevant to cohesion, since they are already accounted for by DS-interactions, and we are interested in evaluating the degree of cohesion between data declarations and subroutines seen as a whole. Also, we do not intend that cohesion should change just because there are two parameters of the same type in a subroutine interface, instead of one of that type. Furthermore, cohesive interactions occur between data declarations and subroutines belonging to the same module. Interactions across different modules are not considered cohesive, since cohesion is the extent to which a module contains data declarations and subroutines that are conceptually related to each other. Interactions across different modules contribute to coupling. Therefore, given a software part *sp*, the sets of cohesive interactions of its constituent modules (if any) are disjoint. In Fig. 3, we show the cohesive interaction graph for the code fragment of Fig. 1.

Interactions across modules when one is a component of another can also be deemed as contributing to cohesion. We did not include these interactions in our evaluation of cohesion; this could be a subject of future research.

It is worth reminding the reader that those relationships that cannot be entirely detected by inspecting the interfaces, i.e., global variables interacting with subroutine bodies, can actually be quite relevant to cohesion evaluation, because they often represent the connections between an object and the subroutines that manipulate it. However, although we expect these unknown interactions to introduce uncertainty in our models, practical experience suggests that the models may still be good, early indicators.

Some care must be used in defining *CI*(*m*) and *M*(*sp*) for languages like Ada that allow circular type definitions, such as the ones used to define the nodes of a linked list. In this case, the declarations of two types *T1* and *T2* are built in such

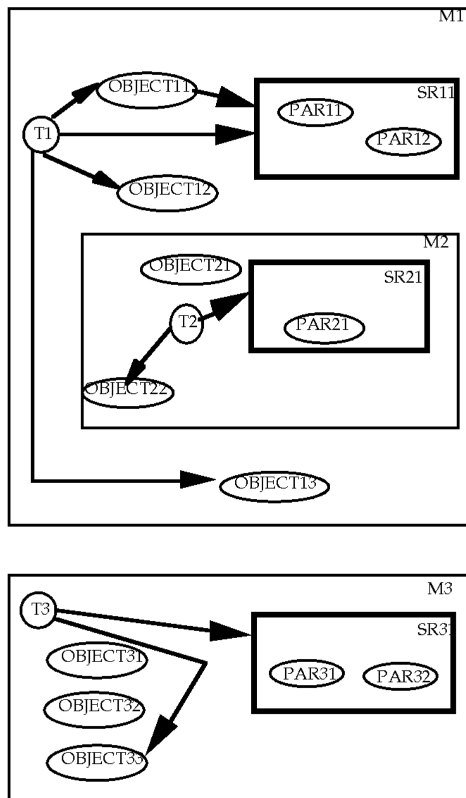


Fig. 3. Cohesive interaction graph for the code fragment of Fig. 1.

a way that *T1* interacts with *T2* and *T2* interacts with *T1*. We choose to include in *CI*(*m*) and *M*(*sp*) only one interaction between them. This is explained by the fact that a single interaction between two data declarations may justify their encapsulation in a single module/Abstract Data Type.

As a result, for a module *m*, we have:

$$|M(m)| = (|DataDeclarations| \cdot (|DataDeclarations| - 1)/2) + |DataDeclarations| \cdot |Subroutines|,$$

where *DataDeclarations* and *Subroutines* are the sets of *m's* data declarations (outside subroutines' interfaces) and subroutines, respectively. $|M(sp)|$ is the sum of all values obtained for $|M(m)|$ for all modules, since all *M*(*m*)*s* are disjoint. For example the maximal number of cohesive interactions for module *M2* of Fig. 1 is *M*(*M2*) = 6.

There are two particular cases in which *M*(*m*) and, therefore, *CI*(*m*) are empty: 1) module *m* contains no data declarations at all (it is either empty or contains only a set of subroutines) or 2) module *m* only contains a single data declaration and no subroutines. In both cases, no cohesive interactions are possible. According to our notion of cohesion, we are interested in the tightness of relationships of data declarations with other data declarations or subroutines, which are supposed to be related to the data declarations. In case 1, there are no data declarations and, therefore, there is a complete absence of cohesion. On the other hand, a single data declaration (case 2) is highly cohesive in itself, so a module only containing one data declaration is highly cohesive. In what follows, given a software part *sp*,

- *SSR*(*sp*) will denote the set of subroutines belonging to modules that do not contain any data declarations (case 1), and
- *SDD*(*sp*) will denote the set of modules of *sp* that only contain a single data declaration and no subroutines (case 2).

4.3 Properties for Interaction-Based Cohesion Measures

We now introduce the following three properties that we believe characterize cohesion measures in our specific Ada context for interaction-based measures.[2] These properties are instantiations, for our specific context, of the properties defined in [12] for cohesion.

Property AdaCohesion.1: Normalization. *Given a software part* sp, *a measure* cohesion_measure(sp) *belongs to a specified interval* [0, Max]. cohesion_measure(sp) = 0 *if and only if* CI(sp) *and* SDD(sp) *are empty, and* cohesion_measure(sp) = Max *if and only if* CI(sp) = M(sp) *and* SSR(sp) *is empty*.[3]

Normalization can provide support for meaningful comparisons between the cohesions of different software parts, since they all belong to the same interval. In addition, the larger the size of a module, the higher the likelihood of a large number of interactions. Normalization helps us make sure our measures are not statistically associated with the size of the modules since it takes into account the potential for a larger number of interactions in large modules.

Property AdaCohesion.2: Monotonicity. *Let* sp_1 *be a software part and* $CI(sp_1)$ *its set of cohesive interactions. If* sp_2 *is a modified version of* sp_1 *with the same sets of data and subroutine declarations and one more cohesive interaction so that* $CI(sp_2)$ *includes* $CI(sp_1)$, *then* $cohesion_measure(sp_2) \geq cohesion_measure(sp_1)$.

Adding cohesive interactions to a software part cannot decrease its cohesion. This is an intuitive property since, if the module's declarations appear to be more interdependent, cohesion should not decrease. For instance, the following program fragment

```
C : constant INTEGER := 100;
type A is array(1..C) of INTEGER;
```

has one more cohesive interaction than

```
C : constant INTEGER := 100;
type A is array(1..100) of INTEGER;
```

Property AdaCohesion.3: Cohesive Modules. *Let* sp *be a software part, and let* m_1 *and* m_2 *be two of its modules. Let* sp' *be the software part obtained from* sp *by merging the declarations belonging to* m_1 *and* m_2 *into a new module* m. *If no cohesive interactions exist between the declarations belonging to* m_1 *and* m_2 *when they are grouped in* m, *then* cohesion_measure(sp) ≥ cohesion_measure(sp').

This property can also be interpreted as follows: Splitting two sets of declarations which are not related to each other via cohesive interactions into two separate modules cannot decrease the cohesion of the software part. Such a property is also intuitively justified since, if two independent modules can be extracted from a module, then there was no reason for them to be merged together in the first place.

2. Properties and measures can be defined for module sets more general than software parts. However, for simplicity, we will provide them only for software parts.

3. We assume that each module contains at least one data declaration or one subroutine, i.e., we will not consider empty modules.

Properties AdaCohesion.1—AdaCohesion.3 are meaningful only for measures defined at the ratio level of measurement [23]. This does not imply that all measures that satisfy them are defined at the ratio level of measurement. Also, it is worth reminding the reader that the fact that the above measures satisfy properties AdaCohesion.1—AdaCohesion.3 should be interpreted as a necessary condition for them to be taken as cohesion measures in our property-based framework, which one can subjectively accept or reject and replace with another one.

4.4 Measure Definition

Based on the previous properties defined, we introduce a measure to capture interaction-based cohesion for software parts in our context.

Measure 1: Ratio of Cohesive Interactions (RCI) for a Software Part.

The Ratio of Cohesive Interactions *for* sp *is*

$$\mathrm{RCI(sp)} = \frac{|\mathrm{SDD(sp)}| + |\mathrm{CI(sp)}|}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp)}|} \quad (1)$$

As an example, with reference to Figs. 1 and 3, *RCI*(*M2*) = 2/6 = 0.333. It can be shown that RCI(sp) satisfies the above properties AdaCohesion.1—AdaCohesion.3, but it does not satisfy any of the sets of properties for size, length, complexity, or coupling defined in [12]. Therefore, since it is consistent with our intuitive and formally defined understanding of cohesion, we believe that *RCI*(*sp*) is a valid cohesion measure in our application context. It is also important to note that these concepts are still very subjectively defined in the software engineering community and that, consequently, there is no real widely accepted reference framework for cohesion that we can use to demonstrate the construct validity of a cohesion measure.

As for the level of measurement of *RCI(sp)*, although this is ultimately a subjective matter that can rarely be formally demonstrated [7], we will interpret *RCI*(*sp*) as a ratio scale measure, based on the following evidence.

1) When $SDD(sp) = \emptyset$, i.e., no module in sp contains only a single data declaration, and $SSR(sp) = \emptyset$, i.e., no module in *sp* contains no data declaration, the value of *RCI*(*sp*) can be computed as

$$RCI(sp) = \frac{|CI(sp)|}{|M(sp)|},$$

and is defined on a ratio scale since this is a ratio of two counts having the same measurement unit. In practical cases—as the ones we show in Section 7—this ratio is very close to that computed by formula (1), since there are few modules that only contain a single data declaration and nothing else, and the number of subroutines in *SSR*(*sp*) is quite small with respect to the maximum number of potential cohesive interactions. Few modules only contain subroutines, and $|SSR(sp)|$ only grows linearly with the number of subroutines in such modules. Instead, $|M(sp)|$ grows quadratically with the number of data declarations and linearly with the number of subroutines in the whole software part. Therefore, in practical situations, it can be shown that formula (1) is approximately at a ratio level of measurement.

2) The usual statistical tests and regression techniques requiring at least interval scale measurement can be safely applied even if a measure is defined on a scale which is only approximately interval [7].

RCI(*sp*) can also be computed as a weighted sum of the *RCI*(*m*)*s* of the single modules m belonging to *sp*. Since cohesive interactions only occur within modules, but not across modules,[4] the numerator of (1) is calculated as

$$|\mathrm{SDD(sp)}| + |\mathrm{CI(sp)}| = \sum_{m\in sp} (|\mathrm{SDD(m)}| + |\mathrm{CI(m)}|)$$

so

$$\mathrm{RCI(sp)} = \frac{\sum_{m\in sp} (|\mathrm{SDD(m)}| + |\mathrm{CI(m)}|)}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp}|} = \sum_{m\in sp} \frac{|\mathrm{SDD(m)}| + |\mathrm{CI(m)}|}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp}|}$$

By multiplying and dividing each term in the summation by $|SDD(m)| + |M(m)| + |SSR(m)|$, we obtain

$$\mathrm{RCI(sp)} = \sum_{m\in sp} \left(\frac{|\mathrm{SDD(m)}| + |\mathrm{M(m)}| + |\mathrm{SSR(m)}|}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp)}|} \cdot \frac{|\mathrm{SDD(m)}| + |\mathrm{CI(m)}|}{|\mathrm{SDD(m)}| + |\mathrm{M(m)}| + |\mathrm{SSR(m)}|}\right) = \sum_{m\in sp} \left(\frac{|\mathrm{SDD(m)}| + |\mathrm{Mm)}| + |\mathrm{SSR(m)}|}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp)}|} \mathrm{RCI(m)}\right)$$

The weights represent the potential contribution of each module *m* belonging to the software part *sp* to the cohesion of the whole *sp*. Therefore, the potential contribution of a module of *SDD*(*sp*) is

$$\frac{1}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp)}|}$$

and that of any other module *m* of *sp* that does not contain only subroutines is[5]

$$\frac{|\mathrm{M(m)}|}{|\mathrm{SDD(sp)}| + |\mathrm{M(sp)}| + |\mathrm{SSR(sp)}|}$$

Based on the above cohesion measure, we can define a threshold that can be used as a support for deciding whether a set of data and subroutines should be kept in one single module or divided into two or more modules. For simplicity, we will show here only the case in which we have to decide whether the declarations belonging to a module *m* should be split into two modules m_1 and m_2, where both $M(m_1)$ and $M(m_2)$ are not empty. This should be the case if the cohesion of the software part consisting of the two modules m_1 and m_2 is greater than the cohesion of module *m*, i.e.,

4. In the following formulae, $|SDD(m)|$ may only take the values 1 (when module *m* only contains a single data declaration and nothing else) or 0 (otherwise).

5. For a module *m* with subroutines and no data declarations, *RCI(m)* = 0.

$$\frac{|\mathrm{CI}(m_1)| + |\mathrm{CI}(m_2)|}{|\mathrm{M}(m_1)| + |\mathrm{M}(m_2)|} > \frac{|\mathrm{CI}(m_1)| + |\mathrm{CI}(m_2)| + |\mathrm{CI}_{12}|}{|\mathrm{M}(m)|}$$

where $|CI_{12}|$ is the number of cohesive interactions between the declarations belonging to modules m_1 and m_2 when they are in module *m*. Based on the above inequality, we can define a threshold on $|CI_{12}|$, as follows

$$\frac{(|\mathrm{M}(m)| - |\mathrm{M}(m_1)| - |\mathrm{M}(m_2)|)(|\mathrm{CI}(m_1)| + |\mathrm{CI}(m_2)|)}{|\mathrm{M}(m_1)| + |\mathrm{M}(m_2)|} > |\mathrm{CI}_{12}|$$

We want to emphasize, however, that, since cohesion is not the only attribute relevant to software design—for instance, coupling and reusability are as important as cohesion,—an increase in cohesion should not be used as the only criterion on which to base such a decision.

4.5 The Role of Additional Information

Additional information to what is visible in the interfaces is usually available at the end of high-level design. For instance, given the interface of a module *m* and assuming that the use of some objects is not specified in the subroutine's interface, the designers have at least a rough idea of which objects declared in *m* will be manipulated by a subroutine that appears in *m*'s interface. It will be left to the person responsible for the measure program to decide whether it is worth collecting this kind of information, thus making the designer describe which objects will be accessed by which subroutines. For instance, from the code fragment in Fig. 1, we cannot tell whether *OBJECT12* DS-interacts with subroutine *SR11*. In this case, designers can answer in three different ways:

1) *OBJECT12* DS-interacts with *SR11*;
2) *OBJECT12* does not DS-interact with *SR11*;
3) the information they have is not sufficient.

It is worth saying that answers of kind case 2 provide valuable, though negative, information on the DS-interactions present in a system. For instance, in the code fragment in Fig. 1, the designer may indicate the existence of a DD-interaction between object *OBJECT12* and *PAR12* and the lack of interaction between *OBJECT21* and *PAR21*. As a consequence, the computation of cohesion is affected. If we take into account this additional information, other alternative cohesion measures can be defined.

Given a software part *sp*, and a pair <*A*, *B*>, where *A* is a data declaration and *B* is either a data declaration or a subroutine, we will say that the interaction between them is known if it is detectable from the high-level design or is signaled by the designers, i.e., they provide an answer similar to answer case 1; we will say that the interaction between them is unknown if it is not detectable from the high-level design and is not signaled by the designers, i.e., they provide an answer similar to answer case 3.

The set of known interactions of a software part *sp* will be denoted by *K*(*sp*), and the set of unknown interactions by *U*(*sp*). In general, $|M(sp)| \geq |K(sp)| + |U(sp)|$, since some interactions may not be detectable from the high-level design and the designers may explicitly exclude their existence, i.e., they provide an answer similar to answer case 2.

Measure 2: Neutral Ratio of Cohesive Interactions (NRCI).

Unknown CIs *are not taken into account.*

$$\mathrm{NRCI}(sp) = \frac{|\mathrm{SDD}(sp)| + |\mathrm{K}(sp)|}{|\mathrm{SDD}(sp)| + |\mathrm{M})(sp)| + |\mathrm{SSR}(sp)| - |\mathrm{U}(sp)|}$$

Measure 3: Pessimistic Ratio of Cohesive Interactions (PRCI).

Unknown CIs *are considered as if they were known not to be actual interactions.*

$$\mathrm{PRCI}(sp) = \frac{|\mathrm{SDD}(sp)| + |\mathrm{K}(sp)|}{|\mathrm{SDD}(sp)| + |\mathrm{M}(sp)| + |\mathrm{SSR}(sp)|}$$

(*This is equivalent to* RCI(sp).)

Measure 4: Optimistic Ratio of Cohesive Interactions (ORCI).

Unknown CIs are considered as if they were known to be actual interactions.

$$\mathrm{ORCI}(sp) = \frac{|\mathrm{SDD}(sp)| + |\mathrm{K}(sp)| + |\mathrm{U}(sp)|}{|\mathrm{SDD}(sp)| + |\mathrm{M}(sp)| + |\mathrm{SSR}(sp)|}$$

The above three measures satisfy properties AdaCohesion.1—AdaCohesion.3, where *CI*(*sp*) is replaced by *K*(*sp*) ∪ *U*(*sp*).

If *PRCI*(*sp*), *NRCI*(*sp*), and *ORCI*(*sp*) are all not undefined,[6] it can be shown that, for all software parts *sp*,

$$0 \leq \mathrm{PRCI}(sp) \leq \mathrm{NRCI}(sp) \leq \mathrm{ORCI}(sp) \leq 1$$

ORCI(*sp*) and *PRCI*(*sp*) provide the bounds of the admissible range for cohesion, and *NRCI*(*sp*) takes a value in between. It can also be shown that the smaller the number of unknown interactions, the smaller the interval [*PRCI, ORCI*], i.e., the more complete the information, the narrower the uncertainty interval. It should be noted that, once the low-level design is completed, accurate and complete information about cohesive interactions should be available.

In addition, *NRCI*(*sp*) is undefined if and only if all interactions are unknown and both *SDD*(*sp*) and *SSR*(*sp*) are empty, i.e., no information is available on cohesion. It is interesting to notice that in this case, and only in this case, *PRCI*(*sp*) = *0* and *ORCI*(*sp*) = 1, i.e., *PRCI*(*sp*) and *ORCI*(*sp*) do not provide stricter bounds than the ones provided by the interval for cohesion. The fact that *NRCI*(*sp*) is undefined can be interpreted as the possibility that *NRCI*(*sp*) can take any value in the interval [0, 1].

4.6 Related Work

As stated in [25], cohesion is an internal property of a module. A module has high cohesion if its elements are strongly related. The intuitive idea behind this is that elements should be grouped together into modules for logical reasons in order

6. *PRCI*(*sp*) and *ORCI*(*sp*) are undefined when $|SDD(sp)| + |M(sp)| + |SSR(sp)| = 0$, i.e., the software part is empty; *NRCI*(*sp*) is undefined when $|SDD(sp)| + |M(sp)| + |SSR(sp)| - |U(sp)| = 0$, i.e., no known interactions exist and both *SDD*(*sp*) and *SSR*(*sp*) are empty.

to achieve common goals. Thus, it is assumed that modular systems with high cohesion are easier to understand and that the reuse of their modules is facilitated. However, the notions of modules, elements, and relations vary according to the context in which cohesion is to be defined.

4.6.1 Procedural Cohesion

In [20], one of the first operational definitions of cohesion was provided. In this context, modules were subroutines and cohesion was measured on an ordinal scale of measurement: functional, sequential, communicational, procedural, temporal, logical, coincidental (in decreasing order of cohesion). The criteria used to define this scale focus on the relationships (or the lack thereof) that exist between the functions embedded in a subroutine [23]:

- Functional cohesion implies that the subroutine performs a single well defined function.
- Sequential cohesion implies that the subroutine's functions are performed in a sequential order described by the subroutine's specifications.
- Communicational cohesion implies that the subroutine's functions are performed on the same body of data.
- Procedural cohesion implies that the subroutine's functions are related to the same general procedure.
- Temporal cohesion implies that the subroutine's functions are related because they must occur within the same time span.
- Logical cohesion implies that the subroutine's functions are only related logically.
- Coincidental cohesion means that none of the relationships mentioned above exist between the subroutine's functions.

As Fenton pointed out in [23], because the trend is now towards languages and methods that support abstract data types (ADTs) encapsulated into modules, e.g., Ada packages, C++ classes, the notion of cohesion should be extended to a higher level and adapted to ADTs where elements are subroutines and declarations. This may seem to contradict the above definition of cohesion (focusing on subroutine cohesion) since ADTs usually contain several subroutines performing different functions which may not be related according to the most important relations underlying the ordinal scale of cohesion. This is discussed below:

- *Sequential cohesion*: subroutines, i.e., methods according to the object-based/object-oriented terminology, in an ADT do not have to be executed in a predetermined order according to the ADT's specifications although *Create_object* and *Destroy_object* methods are, respectively, always the first and last operation on a given object.
- *Communicational cohesion:* subroutines in an ADT usually work, from a general perspective, on the same body of data: the abstract data type itself. However, they may initialize/access/update the values of different attributes, all being elements of the abstract data type. More concretly, an abstract data type may be implemented as a set of distinct data structures all encapsulated in a single module. Subroutines inside that module may work on different subsets of those data structures.
- *Procedural cohesion*: There is no reason for subroutines in an ADT to perform functions belonging to a general procedure. For example, geomeasureal operations, e.g., rotations, translations, may be part of different procedures to manipulate geomeasureal objects, e.g., drawing tools, graphical simulations, etc.
- *Temporal cohesion*: there is no reason for subroutines in an ADT to be executed within the same time span.

Therefore, the basis for encapsulation into modules makes it less likely that one can find some of the forms of cohesion in the [20] classification. However, subroutines and declarations in ADTs should be somewhat related since they should all perform operations on the abstract data type, e.g., push, pop are operations on the ADT Stack, and this may be seen as another form of cohesion. Fenton calls this kind of cohesion "abstract cohesion" and mentions that, unfortunately, there are no obvious measurement procedure and no graph-type model to capture it. In Section 4.4, our goal was to take a step in that direction, to provide a measure of ADT cohesion which is based on the interaction graph model presented above and which can be captured through automatable data collection procedures.

A proposal for functional cohesion measures can be found in [14]. Given a procedure, function, or main program, only *data tokens,* i.e., the occurrence of a definition or use of a variable or a constant, are taken into account. The *data slice* for a data token is the sequence of all those data tokens in the program that can influence the statement in which the data token appears, or can be influenced by that statement. Being a sequence, a data slice is ordered: it lists its data tokens in order of appearance in the procedure, function or main program. If more than one data slice exists, some data tokens may belong to more than one data slice: these are called *glue tokens.* A subset of the glue tokens may belong to all data slices: these are called *superglue tokens.* Functional cohesion measures are defined based on data tokens, glue tokens, and superglue tokens. Given a procedure, function, or main program p, the following measures *SFC*(*p*) (Strong Functional Cohesion), *WFC*(*p*) (Weak Functional Cohesion), and *A*(*p*) (adhesiveness) are introduced.

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

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

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

It can be shown that these measures satisfy the properties defined in [12] for cohesion. However, these measures refer to the functional cohesion of procedures, functions, or main programs based on code-level information. They are, therefore, out of the scope of our study.

4.6.2 Object-Based/Object-Oriented Cohesion

Reference [19] introduced a well-known suite of object-oriented measures and, as such, some of them are also adaptable and applicable to abstract data types. More specifically, a measure for the *lack of cohesion* (*LCOM*) was defined. For a class in an OO design, this is the number of member functions pairs without shared instance variables minus the number of member functions with shared instance variables. However, the measure is set to 0 whenever the above subtraction is negative. In [3], we have shown that *LCOM* is not a significant predictor of fault-prone classes. This could be easily explained since the distribution of *LCOM* showed a lack of variability in the studied sample since most classes had a null *LCOM*. This stems in part from the definition of *LCOM* where the measure is set to 0 when the number of class pairs sharing variable instances is larger than that of the ones not sharing any instances. Several other measures have been proposed in the literature for object-oriented cohesion, e.g., see [8], [28]. Due to space constraints, no thorough comparison can be made here. The interested reader is referred to [6], where an extensive survey and classification have been proposed. From a general perspective, these measures differ according to their underlying experimental hypotheses and properties. At a higher level, several criteria capture the main differences: the types of connections/dependencies that increase cohesion, the domain of the measure, e.g., subroutine, class, set of classes, whether direct or indirect connections are taken into account, how inheritance is handled. In our case, based on our experimental hypotheses, the notion of interaction has been defined to capture the types of dependencies of interest, we define measures for modules and set of modules, we investigate both direct and indirect interactions, and we do not consider inheritance since we work in the context of object-based systems.

5 Interaction-Based Measures for Coupling

In our context, coupling is the extent to which a software part is related to other software parts. We define coupling as a property of an individual software part, or more specifically a relation between an individual software part and its associated software system, rather than as a relation between two software parts as has been done in other contexts [20]. By viewing coupling with respect to an individual part, we are better able to assess the design quality of that part as it relates to the part's fault-proneness. Coupling can be divided into two parts:

1) *import coupling*, i.e., the extent to which a software part depends on the rest of the sotware system, and
2) *export coupling*, i.e., the extent to which the rest of the software system depends on the software part.

Here, we will focus only on import coupling, since our hypotheses for export coupling were not confirmed by our experimental validation. More information on that topic can be found in [10].

In this section, we first give an experimental hypothesis on import coupling, which provides the rationale for our study (Section 5.1). Then, we introduce the abstraction we use for defining our coupling measure (Section 5.2). The instantiation of the coupling properties defined in [12] for our application case is in Section 5.3. An interaction-based measure is defined in Section 5.4. Section 5.5 discusses the issue of genericity in the context of coupling. Related previous works will be presented in Section 5.6.

5.1 Experimental Hypothesis

The following experimental hypothesis provides the motivations for the measure we define.

Hypothesis H-IC: *The more dependent a software part on external data declarations, the more external information needs to be known in order to make the software part consistent with the rest of the system. In other words, the larger the amount of external data declarations, the more incomplete the local description of the software part interface, the more spread the information necessary to integrate a software part in a system. Thus, the software part becomes more fault-prone.*

Like with Hypothesis H-CH, this hypothesis establishes a link between two software attributes: coupling and fault-proneness. This hypothesis is one of the experimental hypotheses believed to be true in our context that our empirical study has confirmed.

We recognize that, in addition to the quantity of external data required, the diversity of other parts in the software system from which external data must be obtained may well contribute to coupling. In Section 6, we will discuss a measure of coupling that tries to capture this diversity.

5.2 Abstraction Definition

Import coupling of a software part will be expressed in terms of the actual DD-interactions involving imported external data declarations and the internal data declarations of the software part. Therefore, the abstraction we use is the DD-interaction graph, of which we will consider only the interactions across software parts.

5.3 Properties for Interaction-Based Coupling Measures

We now provide properties that we believe should be satisfied by interaction-based import coupling measures. These properties are instantiations, for our specific Ada context, of the properties defined in [12] for coupling.

Property AdaCoupling.1: Nonnegativity. *Given a software part* sp, *the measure* import_coupling_measure(sp) ≥ 0. import_coupling_measure(sp) = 0 *if* sp *does not have import interactions with other software parts.*

Property AdaCoupling.2: Monotonicity. *Let* m_1 *be a module and* $II(m_1)$, *its set of import interactions. If* m_2 *is a modified version of* m_1 *with the same sets of data and subroutine declarations and one more import interaction so that* $II(m_2)$ *includes* $II(m_1)$, *then* import_coupling_measure(m_2) ≥ import_coupling_measure(m_1).

Adding import interactions to a module cannot decrease its import coupling.

Property AdaCoupling.3: Merging of Modules. *The sum of the import couplings of two modules is no less than the coupling of the module which is composed of the data declarations of the two modules.*

This stems from the fact that two modules may contain interactions between each other's declarations, which are no longer import interactions for the module resulting from merging the original modules.

It should be noted that, as opposed to cohesion, measures for coupling are not normalized. This comes from hypotheses H-CH and H-IC, where we state that cohesion is a *degree* of interdependence within a software part, whereas coupling is an *amount* of dependencies between a software part and the rest of the system. As with cohesion, properties AdaCoupling.1—AdaCoupling.3 are meaningful for ratio scale measures.

5.4 Measure Definition

We will now introduce interaction-based coupling measures. The issue will be first addressed by ignoring generic modules for the sake of simplification. Generic modules and their impact on the defined measures will be treated in Section 5.5.

Measure 5: Import Coupling

Given a software part sp, *Import Coupling of* sp (*denoted by* IC(sp)) *is the number of DD-interactions between data declarations external to* sp *and the data declarations within* sp.

It can be shown that *IC*(*sp*) satisfies the above properties AdaCoupling.1—AdaCoupling.3, but it does not satisfy any of the sets of properties for size, length, complexity, or cohesion we formally defined in [12]. Therefore, since it is consistent with our intuitive and formally defined understanding of coupling, we believe that *IC*(*sp*) is a valid coupling measure in our application context. As for its measurement scale, *IC*(*sp*) is a count of interactions and may therefore be used as an absolute scale measure, and, as a consequence, as a ratio scale measure.

Each box in Fig. 4 represents a module interface. Module interfaces *m2* and *m3* are located in their parent's interface *m1*. *m2* is assumed to be declared before *m3* and therefore visible to *m3*. T_{ij} and $OBJECT_{ij}$ data declarations represent, respectively, types and objects in module *mi*. *FP3* represents a subroutine formal parameter. The *IC* values for the modules in Fig. 4 are computed as follows:

$$IC(m1) = 0, IC(m2) = 4, IC(m3) = 5, IC(m4) = 2.$$

As visible in Fig. 4, coupling between independent modules is considered in the same way as coupling between modules and submodules. The justification for this is that, when a module *B* is a submodule of a module *A* (*B* IS_COMPONENT_OF *A*), then it implicitly sees part of *A* and explicitly uses some of the declarations of *A*, in the same way as an external module *C* would import, e.g., with clause in Ada, and use declarations from *A*.

Based on the definitions of *IC*(*sp*), we derive two related measures, *DIC*(*sp*) (Direct Import Coupling), *TIC*(*sp*) (Transitive Import Coupling). *DIC*(*sp*) only takes into account direct interactions, whereas *TIC*(*sp*) only takes into account transitive interactions. By their definitions, *IC*(*sp*) = *DIC*(*sp*) + *TIC*(*sp*). This allows us to separately evaluate the impact of direct and transitive interactions on fault-proneness, as we show in the empirical validation. In practice, the number of transitive interactions turns out to be much greater than that of direct interactions, so *IC*(*sp*) ≈ *TIC*(*sp*).

5.5 The Treatment of Generic Modules

There are two possible ways of taking into account generics when calculating coupling. Either each instance can be seen as a different module or a generic can be seen as any other module whose scope/global data declarations is/are the union of the scope/global data declarations of its instances. The second solution does not consider instances as independent modules and appears to be more suitable to our specific perspective, since faults are to be found in generics and, only as a consequence, in instances.

The import coupling of a generic module is the cardinality of the union of the sets of DD-interactions between the data declarations in the software system and those of each of its instances. Consistent with the definition of DD-interaction, generic formal parameters DD-interact with their particular generic actual parameters, i.e., type, object, when the generic module is instantiated, since a change in the former may imply a change in the latter.

This is what the example in Fig. 5 illustrates. *Gen_m* is the interface of a generic module, with a generic formal parameter *GenFP* and a generic type *GenT*. The export coupling of module *Gen_m* is given by the sum of three parts

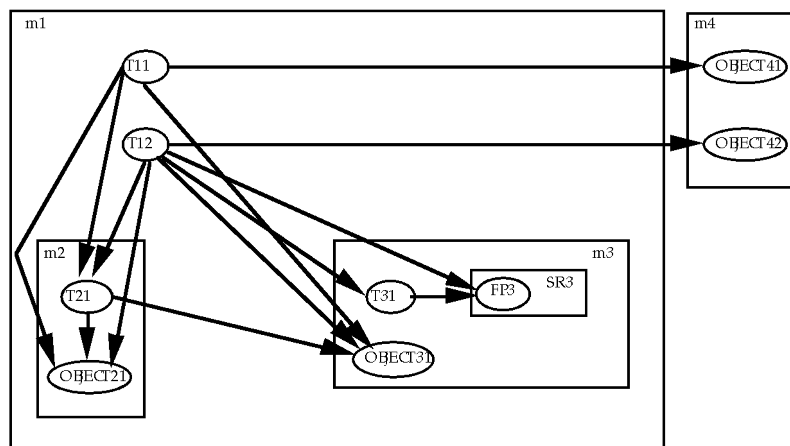


Fig. 4. Calculation of IC with nongeneric modules only.

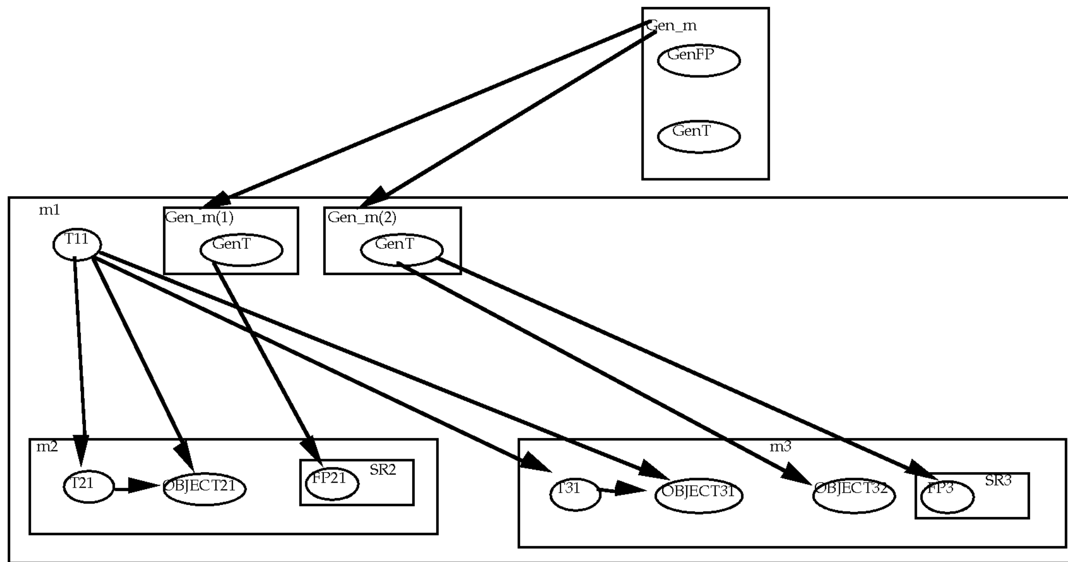


Fig. 5. Generics when calculating import coupling.

1) two interactions from *Gen_m* to *m1*, due to the two instantiations, *Gen_m(1)* and *Gen_m(2)*, of *Gen_m* in *m1*,
2) the interaction from the instantiation *Gen_m(1)*,
3) the two interactions from the instantiation *Gen_m(2)*.

The values of *IC* for the modules in Fig. 5 are as follows:

$$IC(m1) = 2,\ IC(m2) = 3,\ IC(m3) = 4,\ IC(Gen_m) = 0.$$ [7]

5.6 Related Work

As stated in [25], coupling characterizes a module's relationship to other modules and measures their interdependence. Again, it is assumed that low coupling will help analyze, understand, modify, test, and reuse modules separately. Meyer [36] defines the "weak coupling" principle as: if any two modules communicate at all, they should exchange as little information as possible. As with cohesion, the notions of modules, elements, and relations vary according to the context in which coupling is to be defined.

5.6.1 Procedural Coupling

Similarly to cohesion, an ordinal measurement scale was defined for coupling [20] based on whether or not certain relationships occur between subroutines:

- Content coupling
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling

For example, Content coupling occurs when a subroutine refers directly to the inside of another subroutine (e.g., branches into) whereas Common coupling occurs whenever two subroutines refer to the same global data [23]. Content coupling between ADTs is not relevant since it goes against the fundamental definitions of what ADTs are. In addition, based exclusively on high-level design information, it may not always be possible to determine whether or not global data are being shared, or how parameters exchanged between subroutines will be used, e.g., for controlling behavior (Control coupling) or for data exchange (Data coupling). The accuracy of this classification may depend on the high-level design language used. For instance, the knowledge of procedure headers in Ada might not be sufficient by itself to determine how a parameter is used in a procedure. Consistent with the stated objectives of this study, we have provided definitions for ADT coupling in Section 5.4 based on high-level design information formalized through interactions as defined in Section 3.2.

5.6.2 Object-Based/Object-Oriented Coupling

In [19], a measure called Coupling Between Object classes (CBO) has been proposed for classes in object-oriented systems. A class is coupled to another one when it uses its member functions and/or instance variables. CBO provides the number of classes to which a given class is coupled.

In our case, we have chosen to look at the coupling of modules based exclusively on information available at the end of high-level design in our Ada context. We have chosen to look at interaction-level coupling, i.e., at the frequency of interactions between a software part and the others. We think that two interdependent software parts may show very different intensities of interaction and that that should be taken into account.

On the other hand, CBO looks very similar to the ISP measure we will introduce in the next section, except that ISP is defined in our object-based Ada context instead of OO classes.

Many other measures for coupling in object-oriented systems have been provided in the literature (for instance, see [32], [19], [28], [33], [4]—a survey is available in [3]). Comments can be made similar to the above discussion on cohesion. Measures differ according to several criteria and the most important ones are: the types of connection/ dependency contributing to coupling, the locus of impact, i.e., import vs. export coupling, the domain of the measure, its level of granularity, i.e., how connections are counted, and,

7. *Gen_m* may well be coupled to some other module(s), such as those from which the type(s) of its FP come. This coupling is not shown in Fig. 5.

as for cohesion, how indirect connections and inheritance are handled. In our case, certain choices (described above) have been made, based on our experimental hypotheses, regarding these criteria.

6 MEASURES BASED ON USES AND IS_COMPONENT_OF RELATIONSHIPS

These measures are similar to existing measures in the literature [1], [23] and were defined in order to provide a basis of comparison for the measures introduced in the previous sections. Among the ones we investigated [10], two measures appeared to be statistically significant as indicators of fault-proneness and are, therefore, introduced below, *Imported Software Parts*, based on the USES relation among software parts, and *Average Depth* of the nodes of the hierarchy defined by the IS_COMPONENT_OF relations within software parts.

H-ISP is the experimental hypothesis that we believed to be true on the influence of the imported software parts on fault proneness.

Hypothesis H-ISP*. The larger the number of imported software parts, the larger the context to be understood, the more likely the occurrence of a fault.*

Based on this hypothesis, we defined the following measure.

Measure 6: Imported Software Parts.
ISP(sp) *will denote the number of software parts imported and used by a software part* sp.

The relationship we believed to exist between depth of the IS_COMPONENT_OF hierarchy and fault-proneness is expressed by the experimental hypothesis H-A.

Hypothesis H-A*. The larger the depth of a hierarchy, the larger the context information that is available to the lower nodes, the more likely the occurrence of error regarding the hierarchy, the more likely the detection of a fault in it. In other words, if a module* B *is included as a submodule of a module* A *(and not as an independent module, e.g., a library unit in Ada as opposed to a secondary unit), we assume that* B *is not fully understandable out of its context of definition. Otherwise, it would have been defined independently of* A.[8]

This experimental hypothesis allowed us to define the following measure.

Measure 7: Average Depth
Avg_Depth(sp) *will denote the average depth of the nodes composing a software part* sp.

7 EMPIRICAL VALIDATION

In this section, we describe the last step of the approach we have followed in our study, i.e., the empirical validation of the measures we defined. More specifically, we precisely describe the goals of our empirical validation in Section 7.1. In Section 7.2, we show how we have carried out our empirical validation. Sections 7.3, 7.4, 7.5, and 7.6 describe the experimental results we have obtained: Section 7.3 shows the descriptive statistics, Section 7.4 the correlation analysis, and Sections 7.5 and 7.6 discuss the univariate and multivariate analysis results, respectively.

7.1 Goals of the Empirical Validation

In our study, the empirical validation has two main goals.

Goal 1. We want to find out which of the measures defined above have a significant (in the two senses of statistically and practically significant) impact on the fault-proneness of software parts. As said in Section 4.1, fault-proneness is defined in this context as the probability of a fault to be detected in a software part by testing it. We think that such a definition is intuitive and can be handled at low cost in an experimental setting. It also allows us to use a robust and standard modeling technique specifically suitable to classification, i.e., logistic regression [27]. However, other definitions and modeling techniques could be used, e.g., number of faults and least-square regression, respectively.

In this context, we are going to

1) identify which of our high-level design measures are significantly related to software fault-proneness;
2) determine which of our hypotheses are empirically supported;
3) compare the interaction-based strategy to simpler strategies for defining high-level design measures;
4) assess the stability of the observed trends across projects.

Section 7.3, 7.4, and 7.5 show the experimental results related to steps 1, 2, 3, and 4 in Goal 1.

Goal 2. We need to investigate dependencies between measures, in order to determine which ones are complementary and can be used in combination for fault-proneness prediction, and which ones capture similar phenomena and are redundant. In other words, we need to determine whether the defined measures are redundant or complementary explanatory variables of fault-proneness. If they are complementary, then they are all potentially useful in order to build a prediction model for fault-proneness. If most of them are redundant, then a few of those measures are sufficient to help predict fault-proneness, assuming they are significant predictors. We do not expect, though, that our measures explain all of the variation in the data set. We are very well aware that other factors have an impact on fault-proneness, e.g., human factors, code attributes. On the other hand, we want to determine whether they can be a useful part of a prediction model.

Section 7.6 investigates the goodness of fit obtained when building multivariate classification models for detecting fault-prone LMHs based on the design measures that appeared statistically significant during univariate analysis. The model results are assessed and the model structure is investigated.

7.2 Empirical Validation Strategy

In order to validate software measurement hypotheses empirically, one can adopt two main strategies:

8. As indicated in Section 4.2, this can also be a factor in assessing the cohesion of *A*, though we did not include it as a component of our definition of cohesion.

1) small-scale controlled experiments,
2) real-scale industrial case studies.

In this research project, we chose the second alternative since we thought the phenomena we are studying would be even more visible and significant on software systems of realistic size and complexity. Also, we thought that strategy 2 should be a more relevant and convincing validation for the software industry practitioners.

However, the problem in such studies is that it becomes difficult to study the phenomena of interest in isolation, without having to deal with other sources of variation. In our case, we thought that, if these measures were to be interesting, they should explain a significant percentage of the variation individually or in combination, despite other sources of variation. However, we expect some instability across projects.

Environment. The first system studied is an attitude ground support software for satellites (GOADA) developed at the NASA Goddard Space Flight Center. 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 percent of the source code lines have been reused from other systems, verbatim or slightly modified). The third system we studied (TONS) is an onboard navigation system for satellite, 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 percent 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.

During development, change report forms are generated based on testing error reports. These forms contain data on the type, cause, and source of errors. In addition, they provide the modules that are affected by the change. Each module affected is considered to contain a *fault* (following the standard IEEE terminology). Considering that this data collection process has been running and institutionalized for more than 20 years, we expect the data collection to be reliable and complete. No evidence of the contrary was found.

Tool. A tool has been developed to analyze the interface parts of Ada source code, in order to capture the design attributes of these systems. This tool is based on LEX & YACC [34] and extracts generic high-level design information about the visibility and interactions of the system declarations. This information is consequently used to compute the measures presented in Sections 4.4, 5.4, and 6, and others that might be defined.

Analytical Model. The response variable we use to validate the design measures is binary, i.e., Was a fault detected in a LMH or not? In order to analyze the impact of software measures on the fault-proneness of software parts (i.e., probability of a fault to be detected in a software part), we used logistic regression, a classification technique [27] used in many experimental sciences, based on maximum likelihood estimation, and presented below. In particular, we first used univariate logistic regression, to evaluate the impact of each of the measures in isolation on fault-proneness. In this case, a careful outlier analysis must be performed in order to make sure that the observed trend is not the result of few observations [21].[9] Then, we performed multivariate logistic regression, to evaluate the relative impact of those measures that had been assessed sufficiently significant in the univariate analysis. For instance, according to [27], $p < 0.25$, where p is the probability for the regression coefficient to be different from 0 by chance, is a reasonable heuristic to select candidate covariates for multivariate analysis. This modeling process is further described in [27].

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is a special case of this, where only one variable appears):

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{(C_0 + C_1 \bullet X_1 + \ldots + C_n \bullet X_n)}}{1 + e^{(C_0 + C_1 \bullet X_1 + \ldots + C_n \bullet X_n)}} \quad (2)$$

where

- π is the probability that no fault was found in a software part during the validation phase
- the X_is are the design measures included as explanatory variables in the model (called *covariates* of the logistic regression equation).

The curve between π and any single X_i—i.e., assuming that all other X_js are constant—takes a flexible S shape which ranges between two extreme cases:

1) when a variable is not significant, then the curve approximates a horizontal line, i.e., π does not depend on X_i
2) when a variable entirely differentiates fault-prone software parts, then the curve approximates a vertical line.

The coefficients C_is will be estimated through the maximization of a likelihood function, built in the usual fashion, i.e., as the product of the probabilities of the single observations, which are functions of the covariates (whose values are known in the observations) and the coefficients (which are the unknowns). For mathematical convenience, $l = ln[L]$, the loglikelihood, is usually the function to be maximized. This procedure assumes that all observations are statistically independent. In our context, an observation is the detection/non detection of a fault in a LMH. Each detection/nondetection of a fault is assumed to be an event independent from the other fault detections/non detections. This is in part justified by the fact that faults correspond to different change report forms and, therefore, error detection events.

The global measure of goodness of fit we will use for such a model is assessed via R^2—not to be confused with the 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 explanatory variables, the more accurate the model. However, as opposed to the R^2 of

9. In addition, in order to confirm the obtained results, we used nonparametric tests for rank distributions such as the Mann-Whitney U test [18]. Results appeared to be consistent across techniques.

least-square regression, high R^2s are rare for logistic regression. (The interested reader may refer to [27] for a detailed discussion of this issue.) R^2 is defined by the following ratio:

$$R^2 = \frac{LL_S - LL}{LL_S}$$

where

- LL is the loglikelihood obtained by Maximum Likelihood Estimation of the model described in formula (2)
- LL_S is the loglikelihood obtained by Maximum Likelihood Estimation of a model without any variables, i.e., with only C_0. By carrying out all the calculations, it can be shown that LL_S is given by

$$LL_S = m_0 \ln\left(\frac{m_0}{m_0 + m_1}\right) + m_1 \ln\left(\frac{m_1}{m_0 + m_1}\right)$$

where m_0 (respectively, m_1) represents the number of observations for which there are no faults (respectively, there is a fault). Looking at the above formula, $LL_S/(m_0 + m_1)$ may be interpreted as the uncertainty associated with the distribution of the binary dependent variable (no fault detected in a LMH, one fault detected in a LMH), according to Information Theory concepts. It is the uncertainty left when the variableless model is used. Likewise, $LL/(m_0 + m_1)$ may be interpreted as the uncertainty left when the model with the covariates is used. As a consequence, $(LL_S - LL)/(m_0 + m_1)$ may be interpreted as the part of uncertainty that is explained by the model. Therefore, the ratio $(LL_S - LL)/LL_S$ may be interpreted as the proportion of uncertainty explained by the model.

Tables 1, 2, 3, 4, 5, and 6 contain the results we obtained through, respectively, univariate and multivariate logistic regression on the three systems. For each measure, we provide the following statistics:

- C (appearing in Tables 3 and 4), the estimated regression coefficient. The larger the absolute value of the coefficient, the stronger the impact of the covariate on the probability p.
- $\Delta\psi$ (appearing in Table 3 only, i.e., in univariate analysis), which is based on the notion of odds ratio [27], and provides an evaluation of the impact of the measure on the dependent variable. More specifically, the odds ratio $\psi(X_i)$ represents the ratio between the probability of not having a fault and the probability of having a fault when the value of the measure is X_i. As an example, if, for a given value X_i, $\psi(X_i)$ is 2, then it is twice as likely that the software part does not contain faults than that it does contain faults. For each variable X_i, the value of $\Delta\psi_i$ for logistic regression is computed by means of the following formula

$$\Delta\psi = \frac{\psi(X+1)}{\psi(X)}$$

Therefore, $\Delta\psi_i$ represents the reduction/increase in the odds ratio when the value X_i of the measure increases by 1 unit and has the useful property to be independent of X_i in the context of logistic regression. This provides a more intuitive insight than regression coefficients into the impact of explanatory variables. (Since the whole range of RCI is [0, 1], we used 0.01 as the quantum for RCI increase with respect to which $\Delta\psi_{RCI}$ is computed.)
- p (appearing in both tables), the statistical significance of C, which provides an insight into the accuracy of the coefficient estimates. The level of significance of the logistic regression coefficients tells the reader about the probability that the coefficient is different from zero by chance. Historically, a significance threshold (α) of $\alpha = 0.05$, i.e., 5 percent probability, has often been used in univariate analysis to determine whether a variable is a significant predictor. However, the choice of a particular level of significance is ultimately a subjective decision and other levels such as 0.01 or 0.1 are commonly used. The larger the level of significance, the larger the standard deviation of the estimated coefficients, the less believable the calculated impact of the coefficient. The significance test is based on a likelihood ratio test [27] commonly used in the framework of logistic regression.

7.3 Descriptive Statistics

Table 1 presents the descriptive statistics for the three projects we analyze. The minimum, maximum, median, mean, and standard deviation are provided in each table cell for each project. These descriptive statistics will be useful later on when we explain the differences observed in the analysis between the projects. Also, in future replications of this study, comparisons will be made easier if the sample statistics can be compared.

From Table 1, a few strong variations between TONS and the other two projects are visible. The standard deviation, mean and median of *TIC* are smaller for TONS. This may be due to the significant difference in size between the systems and results in fewer transitive interactions in TONS. With respect to *ISP*, differences can be observed between projects' means and standard deviations where TONS shows the largest mean of imported software parts and GOESIM the smallest one.

These differences may have numerous causes. GOADA and GOESIM are older projects and among the earlier Ada developments in the studied environment whereas TONS is a much more recent project. Higher module imports may be due to an increase in complexity over time of the systems developed in the studied environment or to the difference in application domain. Similarly, GOADA shows a much smaller median with respect to cohesion. Considering that GOADA was the first Ada project using object-oriented design in that environment, this circumstance may be explained by a lack of experience with that new technology and its underlying concepts.

7.4 Correlation Analysis

Table 2 presents the computed Pearson's correlation coefficients (R) between the design measures computed for each of the three projects.

TABLE 1
DESCRIPTIVE STATISTICS

Measure	Project	minimum	maximum	median	mean	std dev
ISP	GOADA	0	15	1	1.41	1.65
	GOESIM	0	6	1	1.19	1.13
	TONS	0	18	1	1.69	2.2
Avg_Depth	GOADA	1	2.87	1.75	1.5	0.41
	GOESIM	1	2.86	1.8	1.5	0.43
	TONS	1	1.96	1.67	1.52	0.38
RCI	GOADA	0	1	0.003	0.11	0.17
	GOESIM	0	1	0.083	0.16	0.20
	TONS	0	1	0.034	0.16	0.24
TIC	GOADA	0	172	15.5	30.4	32.7
	GOESIM	0	126	46.0	37.2	32.5
	TONS	0	125	3	8.04	16.8
DIC	GOADA	0	67	3	5.02	9.06
	GOESIM	0	32	3	4.63	6.08
	TONS	0	36	3	5.34	7.22

TABLE 2
LINEAR CORRELATION COEFFICIENTS

Measure	Project	ISP	Avg_Depth	RCI	TIC	DIC
ISP	GOADA	1	-0.05	-0.1	**0.4**	**0.53**
	GOESIM	1	0.08	-0.12	**0.3**	**0.52**
	TONS	1	0.2	-0.1	**0.8**	**0.48**
Avg_Depth	GOADA		1	**-0.4**	0.05	0.18
	GOESIM		1	**-0.4**	-0.08	0.06
	TONS		1	-0.23	0.3	0.38
RCI	GOADA			1	0.02	0.12
	GOESIM			1	-0.24	0.02
	TONS			1	-0.1	-0.2
TIC	GOADA				1	0.4
	GOESIM				1	0.17
	TONS				1	**0.71**
DIC	GOADA					1
	GOESIM					1
	TONS					1

Most of the correlations in Table 2 are weak (the significant ones, at the 0.01 level, are in boldface). *ISP* appears to be significantly correlated to *DIC* across the three projects. However, the relationship is relatively weak. The correlation between *TIC* and *ISP* for TONS is mainly due to an outlier. On the other hand, the correlation between *DIC* and *TIC* is actually stronger ($R = 0.87$) when removing that outlier. A careful analysis of Table 2 allows us to conclude that, in most cases, the five measures presented capture different dimensions in our environment. In other words, they are likely not to be redundant from a predictive point of view. The existing significant correlations will have, however, to be considered in the following analysis.

7.5 Univariate Analysis

In this section, we present the results obtained when analyzing the individual impact of the defined design measures on fault-proneness. Table 3 presents these results by providing the computed regression coefficient (*C*), the variation in odds ratio when increasing the measure's value of a unit ($\Delta\psi$), and the actual statistical significance of *C* (*p*). The number of LMH's of the systems for GOADA, GOESIM, and TONS are 131, 85, 83, respectively.

Results. The best univariate logistic regression R^2s (our measure of goodness of fit) are obtained with the measure *Avg_Depth*: GOADA: $R^2 = 0.115$, GOESIM: $R^2 = 0.14$, and TONS: $R^2 = 0.16$.

Detailed Discussion. Across the three systems under study, regression coefficients show the expected signs and seem to support our hypotheses. For example, *RCI* shows a positive sign and, therefore, suggests that the probability of having no fault detected increases with *RCI*. However, *TIC* and *DIC* do not appear to be very significant in TONS ($p = 0.11$ and 0.08, respectively), whereas they are very significant in the other two systems. The analysis of the distribution of *TIC* in all three systems, respectively, shows that its standard deviation, mean, and median are much smaller in TONS (see Table 1). As a consequence, any trend related to *TIC* may not be visible in the TONS dataset. Since TONS is a significantly

TABLE 3
UNIVARIATE ANALYSIS

Measure	Project	C	Δψ (%)	p
ISP	GOADA	-0.8	45	0.000
	GOESIM	-0.717	49	0.002
	TONS	-0.96	38	0.000
Avg_Depth	GOADA	-2.27	11	0.000
	GOESIM	-2.4	9	0.000
	TONS	-3.9	2	0.000
RCI	GOADA	0.63	19	0.000
	GOESIM	0.215	12	0.047
	TONS	0.34	14	0.001
TIC	GOADA	-0.016	98	0.001
	GOESIM	-0.017	98	0.002
	TONS	-0.03	96	0.08
DIC	GOADA	-0.23	79	0.000
	GOESIM	-0.19	83	0.001
	TONS	-0.05	95	0.11

smaller system than the other two, we can hypothesize the following possible explanation: the distribution of indirect import interactions is strongly dependent on the size of the system and indirect import interaction measures are likely to be mediocre predictors for small systems. However, well-founded interpretations of this experimental result require more thorough and extensive studies, based on a larger set of software systems. Other interpretations might turn out to be as plausible. With respect to *DIC*, no explanation has been found for its mediocre level of significance in TONS. In all other cases, the univariate analysis results show that the five defined measures are significantly related (at a $\alpha = 0.05$ level of significance) to fault-proneness.

Comparing Models. Variations across models. i.e., univariate regression equations, should be expected, due to differences in project characteristics and measure distributions, i.e., size, application domain. In order to evaluate the stability of the models, the reader should look at the $\Delta\psi$ columns in Table 3. Model stability may be defined as the degree of variation of the Δψs across projects. Based on that definition, it is worth noticing that, despite the fact that these projects belong to different application domains (within the context of satellite support systems) and have been developed at different times, most of the models are surprisingly stable across projects, i.e., trends are similar and percentages are in similar ranges.

As a conclusion, Goal 1 of our empirical validation is fulfilled by the above analysis since some high-level design measures are significantly related to fault-proneness (see *C* and *p* values in Table 3) (subgoal a). In addition, by analyzing the trends indicated by the coefficients, we see that the hypotheses underlying the measures identified above as significant are empirically supported (subgoal b). Interaction-based measures do not appear to be strongly associated with simpler high-level design measures (Table 2) and, therefore, seem to be complementary (subgoal c). Last, the observed trends appear stable across projects (Δψs in Table 3) (subgoal d).

7.6 Multivariate Models

In this section, we present the results obtained by performing a stepwise *multivariate* logistic regression. Table 4 provides the estimated regression coefficients (*C*) and their significance (*p*) based on a *likelihood ratio test* [27], which is obtained by comparing the maximum likelihood estimate of a parameter to its estimated standard deviation. Regression coefficients are not shown when their level of significance is above 0.25 (substituted by a *).

It is important to note that we do not expect high-level design measures to account for all of the variation of fault-proneness, since other factors are likely to be important too, e.g., human factors. However, the goal of multivariate analysis here is to determine whether the measures appearing significant in the univariate analysis are *complementary* and *useful* for prediction, i.e., useful to build a classifier. In order to do so, we have to show that these measures are,

TABLE 4
COEFFICIENTS OF MULTIVARIATE MODELS

	Project	C	p
ISP	GOADA	-0.9	0.04
	GOESIM	*	*
	TONS	-1.18	0.000
Avg_Depth	GOADA	-1.8	0.003
	GOESIM	-3.12	0.000
	TONS	-5.62	0.000
RCI	GOADA	0.4	0.006
	GOESIM	0.3	0.07
	TONS	0.2	0.16
TIC	GOADA	-0.023	0.000
	GOESIM	-0.02	0.005
	TONS	*	*
DIC	GOADA	0.23	0.04
	GOESIM	-0.13	0.04
	TONS	-0.11	0.002

when used together in a multivariate model, significantly related to fault-proneness. In other words, when measures remain significant covariates when included in the multivariate model, this means that they are *complementary* in explaining fault-proneness. When the multivariate models show a better fit than univariate models, then the measures are deemed to be potentially *useful* for building a multivariate model predicting fault-proneness.

Analyzing the behavior of our measures in a multivariate context allows us to refine their validation by determining the extent to which they can be useful predictors. A detailed discussion of multivariate analysis and the issues mentioned above can be found in [21].

Results. The very low levels of significance (p-values) in Table 4 suggest that, most of the time, these measures may be used in combination as indicators of fault-prone LMHs. Indeed, when used in a multivariate model, many of these measures are still significant and produce models that are more accurate than univariate models (Table 2).

The multivariate R^2s are 0.21 for GOADA, 0.24 for GOESIM, and 0.43 for TONS. These values are, respectively 183, 171, and 269 percent of the best univariate R^2, i.e., the results improved significantly with the multivariate model. (Recall that logistic regression R^2 values are usually low as compared to least-square regression R^2s.)

Interaction-based measures are more complex than *ISP* and *Avg_Depth* but they are worth collecting, since they provide information which is complementary to that provided by *ISP* and *Avg_Depth*. We would miss substantial information if we used only *ISP* and *Avg_Depth*, even though *ISP* and *Avg_Depth* individually perform better than our interaction-based measures. Interaction-level measures allow the building of multivariate models, with better goodness of fit than univariate models. We also want to remark that no other declaration measures we also investigated, e.g., the number of data declarations as a size measure for LMH, turned out to be statistically significant. In addition, the average LMH depth was consistently selected as a very good indicator. *ISP*, a measure similar to the notion of fan-in, shows to be significant across projects (except in the multivariate GOESIM model for reasons explained below). From a more general perspective, measures based on imports, regardless of the associated concepts, appear to explain part of the fault-proneness of software parts.

Comparing Models. Some variability in the estimated regression coefficients can be observed across projects in Table 4. In multivariate models, coefficients have a tendency to adjust, statistically, for other covariates [27], [21]. Sometimes, covariates are weak predictors of the response (or dependent) variable when taken individually, and become more significant when integrated in a multivariate model. In Table 3, *DIC* showed, for TONS, a mediocre level of significance, whereas it appears to be a significant covariate in Table 4. Moreover, its trend is reversed (positive) for GOADA. When looking more carefully at the associations between measures, it can be determined that this may be the results of a significant association between *DIC* and *ISP* (see Table 1) in GOADA. These associations are a typical source of coefficient instability, e.g., the coefficient of *ISP* in GOADA varies from –0.9 to –0.39 when *DIC* is removed from the equation.

TIC does not appear significant in TONS and this may stem from its distribution in *TONS* (Table 1) which shows a much smaller mean and standard deviation in the TONS dataset. If most of TONS's observations lie in the lower range of the *TIC* scale, its impact on fault-proneness may not be visible since we expect LMH's with larger *TIC* values to be fault-prone. Another possible cause is the linear association between *TIC* and *DIC* in TONS (see Table 2). Also, *RCI* does not appear very significant in TONS. In that case, despite the fact that no differences in distribution or strong linear association can be observed (Tables 1 and 2), a strong nonlinear association exists between *RCI* and *DIC*. When using the natural logarithm to transform the scales of *DIC* and *RCI*, i.e., linearize the relationship between *DIC* and *RCI*, a correlation of $R = 0.87$ can be observed. This may very well explain the low level of significance of *RCI* in TONS.

ISP shows a smaller mean and standard deviation in GOESIM and does not appear significant as a covariate in that case. *RCI* shows a level of significance in GOESIM which is worse than in GOADA but better than in TONS. In that case, again, this may be explained by a weak but significant nonlinear relationship between *DIC* and *RCI* (after linearization, $R = 0.46$ or $R = 0.59$ when removing an outlier).

It is important to note that a different set of systems showing different distributions might show very different trends. This points out a need for large scale investigation across various development environments and application domains. In addition, an investigation over a large number of systems would allow us to better determine the ranges of values in which the various measures are significant predictors of fault-proneness.

As a conclusion, Goal 2 of our empirical validation is fulfilled since we have shown that these measures are complementary and useful explanatory variables of fault-proneness, i.e., multivariate models show a better goodness of fit than the univariate models.

Goodness of Fit. In order to better assess the goodness of fit of the above multivariate models, we look now at other measures of fit which provide a perspective complementary to R^2. Let us assume we wish to use the constructed logistic regression models to classify LMH's in two categories, i.e., it is/is not likely to detect a fault in the LMH. In order to do so, we define a probability threshold of 0.5 to decide, based on the computed probability to detect a fault in each LMH, whether a LMH actually contains a fault. In that case, one may decide, for instance, to inspect or test more carefully the LMH.

Based on such classification models, we obtain the classification results presented in Table 5 across the three projects. GOADA, GOESIM, and TONS contain, respectively, 131, 85, and 83 LMHs. In addition, 270, 141, and 115 faults have been reported, respectively. Table 5's rows represent the actual categories of LMHs, i.e., faulty or nonfaulty, whereas the columns represent the classification performed based on the logistic regression models.

TABLE 5
CLASSIFICATION RESULTS FOR THE MULTIVARIATE ANALYSIS

Actual	Predicted					
	GOADA		GOESIM		GOESIM	
	No Fault	Fault	No Fault	Fault	No Fault	Fault
No Fault	22	41	12	33	30	20
Fault	8 (11)	60 (259)	2(11)	38(130)	4(7)	29(108)

Table 6 compares the LMH's which actually contain a fault with the ones that have been predicted to contain a fault. Based on these results, Table 6 presents two classification evaluation criteria: completeness, correctness. The former gives the percentage of faulty LMH's which have been classified as faulty. The latter gives the percentage of times a LMH has been classified correctly as faulty. For example, in GOADA, 60 faulty LMH's have been classified as faulty whereas eight of them have been classified as non-faulty. On the other hand, 41 nonfaulty LMH's have been classified as faulty. In that case, *Completeness* = 60/68 = 88 percent, whereas *Correctness* = 60/101 = 60 percent. These two criteria are complementary in assessing classification results and cannot be analyzed independently. More balanced completeness and correctness results could be obtained by using a different classification threshold than 0.5.

Results put between parentheses in Tables 5 and 6 provide the number of faults detected in each LMH. This allows us to determine correctness and completeness in terms of faults, instead of faulty LMHs. If we take the same example again, *Correctness* = 259/300 = 86 percent and *Completeness* = 259/270 = 96 percent. Overall, the results appear to be substantially better when considering faults. This shows that the models are more accurate for LMHs containing a larger number of faults.

As expected and discussed above, the classification results are not fully satisfactory and there is room for improvement. However, especially with respect to completeness, the results show that the defined design measures are useful indicators of fault-prone LMHs. Furthermore, the design measures show to be excellent predictors of where most of the faults will be detected. Criteria such as correctness and completeness are dependent on the choice of a subjective classification threshold. However, as a measure of fit, they are more intuitive than R^2.

Another important point is that there is a difference between measuring the goodness of fit of a model and assessing its predictive capability. In the latter case, one should define a separate modeling and test sets. The modeling set is usually larger than the test set and is used to build the model. It should be representative of the whole statistical population under study. The test set is used to test the predictive accuracy of the model generated. In our study, we were interested in the goodness of fit and we did not investigate the predictive capability of the model per se. However, a satisfactory goodness of fit is required in order to realistically expect a satisfactory predictive capability in future studies. Our goal was to validate our measures according to the goals stated above, not to build and assess predictive models. Such a task is however a part of our future work and would require larger data sets which are representative of the project population in our environment.

TABLE 6
CLASSIFICATION ACCURACY FOR FAULTY LMHS
FOR THE MULTIVARIATEANALYSIS

	GOADA (%)	GOESIM (%)	GOESIM (%)
Completeness	88 (96)	95 (92)	89 (94)
Correctness	60 (86)	54 (80)	59 (84)

8 CONCLUSION

This paper has presented a methodology and an empirical study on the definition and validation of measures for high-level object-based designs. Our experimental goal was to evaluate the influence of some attributes of the high-level object-based design on the fault-proneness of the produced software in the context of Ada development at NASA/ FSC. Based on the experimental goal, we have set experimental hypotheses from which we derived measures, which were theoretically validated by means of a property-based approach and empirically validated on three real-life software projects.

The study has shown that statistical models of good statistical significance can be built based on *high-level* design information for systems designed based on abstract data types. In particular, we have identified some early indicators for fault-prone software that may be interpreted as cohesion and coupling measures. The stability of the impact of these measures across projects allows us to draw optimistic conclusions about the use of such quality indicators. In a given application domain, the impact of the defined high-level design measures seems to be relatively stable across projects. When differences appear across projects (especially in the multivariate models), they can be explained either by associations between covariates or by differences in distributions across projects. Using early quality indicators based on objective empirical evidence is therefore a realistic objective. Quality indicators can be weighted according to their impact on fault-proneness in order to build quality models and these weights will be representative, to some extent, across projects in the given application domain. However, there is no guarantee that these kinds of indicators will behave similarly across various application domains and development environments. Therefore, it is generally prudent to precede the use of such indicators by a careful empirical analysis of local fault patterns in the studied environment and a thorough comparison across projects. As discussed in the Introduction, we do not believe that universally valid quality measures and models can be

devised at this stage. Therefore, our approach to measure definition and validation can be reused, but the measures and models themselves should be investigated and validated locally in each studied environment.

Our future work will be four-fold to:

1) analyze more systems.
2) assess, as objectively as possible, the predictive capability of models based on high-level design measures.
3) further validate and refine the measures we defined in this paper. The variations across environments and the study/comparison of different architectures is likely to give us interesting insights.
4) be consistent with our current objectives, we will address the issues related to building measure-based empirical models earlier in the life cycle. In particular, the next stage of this research will focus on defining and validating measures for formal specifications [9].

Acknowledgments

We would like to thank the anonymous reviewers and Stuart H. Zweben for their thorough review of this paper. Also, we thank Giuseppe Calavaro, Khaled El-Emam, and Chris Lott for their helpful comments on earlier drafts of this paper. This work was supported, in part, by NASA grant NSG-5123 and UMIACS; and the National Science Foundation under grant 01-5-24845. Sandro Morasca's work was carried out under the financial support of CNR and of the Ministero dell' Universitá e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications."

References

[1] W. Agresti and W. Evanco, "Projecting Software Defects from Analyzing Ada Designs," *IEEE Trans. Software Eng.*, vol. 18, no. 11, Nov. 1992.

[2] L.C. Briand, V.R. Basili, and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components," *IEEE Trans. Software Eng.*, vol. 19, no. 11, Nov. 1993.

[3] V.R. Basili, L.C. Briand, and W. Melo, "A Validation of Object-Oriented Measures," *IEEE Trans. Software Eng.*, vol. 22, no. 10, Oct. 1996.

[4] L.C. Briand, P. Devanbu, and W. Melo, "Defining and Validating Design Coupling Measures in Object-Oriented Systems," *Proc. ICSE'97*, Boston.

[5] L.C. Briand, J. Daly, and J. Wuest, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.*, vol. 25, no. 1, pp. 91–121, 1999. Also available at http://www. iese.fhg.de/ISERN/pub/isern.biblio.html

[6] L.C. Briand, J. Daly, and J. Wuest, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.: An Int'l J.*, vol. 3, no. 1, pp. 65–117, 1998. Also available at http://www. iese.fhg.de/ISERN/pub/isern.biblio.html.

[7] L.C. Briand, K. El Emam, and S. Morasca, "On the Application of Measurement Theory to Software Engineering," *Empirical Software Eng.: An Int'l J.*, vol. 1, no. 1, 1996.

[8] J.M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," *Proc. ACM Symp. Software Reusability* (*SSR'94*), pp. 259-262, 1995.

[9] L.C. Briand and S. Morasca, "Software Measurement and Formal Methods: A Case Study Centered on TRIO + Specifications," ICFEM'97, Hiroshima, Japan, Nov. 1997.

[10] L.C. Briand, S. Morasca, and V.R. Basili, "Defining and Validating High-Level Design Measures," CS-TR-3301, Version 1, Univ. of Maryland, College Park, 1994.

[11] L.C. Briand, S. Morasca, and V.R. Basili, "Goal-Driven Definition of Product Metrics Based on Properties," CS-TR-3346, Version 1, Univ. of Maryland, College Park, 1994.

[12] L.C. Briand, S. Morasca, and V.R. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 68-86, Jan. 1996.

[13] G. Booch, *Software Engineering with Ada.* Menlo, Calif.: Benjamin/Cumming, 1987.

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

[15] V.R. Basili and H. Rombach,"The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, vol. 14, no. 6, June, 1988.

[16] V.R Basili, D. Rombach, J. Bailey, and A. Delis, "Ada Reusability and Measurement," CS-TR-2478, Univ. of Maryland, College Park, May 1990.

[17] L.C. Briand, W. Thomas, and C. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Int'l Conf. Software Eng.*, Maryland, May 1993.

[18] J. Capon, *Statistics for the Social Sciences.* Wadworth Publishing, 1988.

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

[20] L. Constantine and E. Yourdon, *Structured Design.* Prentice Hall, 1979.

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

[22] ANSI/MIL-STD-1815A-1983, *Reference Manual of the Ada Programming Languages.* U.S. Department of Defense, 1983.

[23] N.E. Fenton, *Software Measures, A Rigorous Approach.* Chapman & Hall, 1991.

[24] J. Gannon, E. Katz, and V.R. Basili, "Measures for Ada Packages: An Initial Study," *Comm. ACM*, vol. 29, no. 7, July 1986.

[25] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering.* Englewood Cliffs, N.J.: Prentice Hall, 1992.

[26] S. Henry and D. Kafura, "The Evaluation of Systems' Structure Using Quantitative Measures," *Software Practice and Experience*, vol. 14, no. 6, June, 1984.

[27] D. Hosmer and S. Lemeshow, *Applied Logistic Regression.* Wiley-Interscience, 1989.

[28] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *Proc. Int'l Symp. Applied Corporate Computing*, Oct. 1995.

[29] HOOD Technical Group, B. Delatte, M. Heitz, and J. Muller eds., *HOOD Reference Manual.* Prentice Hall, 1993.

[30] D. Ince and M. Shepperd, "System Design Measures: A Review and Perspective," *Proc. Software Eng. '88*, pp. 23-27, 1988.

[31] C.M. Judd, E.R. Smith, and L.H. Kidder, "Research,, *Methods in Social Relations.* Harcourt Brace Jovanovich College Publishers, 1991.

[32] W. Li and S. Henry, "Object-Oriented Metrics That Predict Maintainability," *J. Systems and Software*, vol. 23, n. 2, pp. 111-122, Nov. 1993.

[33] Y. Lee, B. Liang, S. Wu, and F. Wang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," *Proc. Int'l Conf. Software Quali*ty, pp. 81-90, 1995.

[34] J. Levine, T. Mason, and D. Brown, *LEX & YACC.* O'Reilly & Assoc. Inc., 1992.

[35] J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," *SIGPLAN Notices*, vol. 12, no. 10, pp. 61-64, 1977.

[36] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall, 1988.

[37] A. Melton, D. Gustafson, J. Bieman, and A. Baker, "A Mathematical Perspective for Software Measures Research," *Software Eng. J.*, Sept. 1990.

[38] H.D. Rombach, "A Controlled Experiment on the Impact of Software Structure and Maintainability," *IEEE Trans. Software Eng.*, vol. 13, no. 5, May, 1987.

[39] H.D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software,* Mar. 1990.

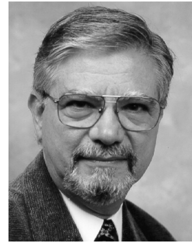
[40] M. Shepperd, "Design Measures: An Empirical Analysis," *Software Eng. J.*, Jan. 1990.

[41] R. Selby and V.R. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.*, vol. 17, no. 2, Feb., 1991.

[42] E.J. Weyuker: “Evaluating Software Complexity Measures,” IEEE *Trans. on Software Eng.,* vol. 14, no. 9, pp. 1,357-1,365, Sept. 1988.
[43] W. Zage, D. Zage, P. McDaniel, and I. Khan, “Evaluating Design Measures on Large-Scale Software,” SERC-TR-106-P, Sept. 1991.
[44] S.H. Zweben, S. Edwards, B. Weide, and J. Hollingsworth, “The Effects of Layering and Encapsulation on Software Development Cost and Quality,” *IEEE Trans. Software Eng.*, vol. 21, no. 3, Mar. 1995.

Lionel C. Briand received the BS degree in geophysics and the MS degree in computer science from the University of Paris VI, France. He received the PhD degree (with high honors) in computer science from the University of Paris XI, France. Dr. Briand is currently head of the Quality and Process Engineering Department at the Fraunhofer Institute for Experimental Software Engineering (FhG IESE), an industry-oriented research center located in Rheinland-Pfalz, Germany. His current research interests and industrial activities include measurement and modeling of software development products and processes, software quality assurance, domain specific architectures, reuse, and reengineering. He has published numerous articles in international conferences and journals and has been a Program Committee member or chair at several conferences such as ICSE, ICSM, ISSRE, METRICS, and SEKE. Before that, he started his career as a software engineer at CISI Ingénerie, France. He then joined, as a research scientist, the NASA Software Engineering Laboratory, a research consortium: NASA Goddard Space Flight Center, University of Maryland, and Computer Science Corporation. Before going to FhG IESE, he held the position of lead researcher in the software engineering group at the Computer Research Institute of Montreal (CRIM), Canada.

Sandro Morasca received his DrEng degree (Italian Laurea) cum laude and his PhD degree in computer science, both from the Politecnico di Milano in 1985 and 1991, respectively. He is currently an associate professor of computer science in the Dipartimento di Elettronica e Informazione at the Politecnico di Milano, where he held the position of assistant professor from 1993–1998. From 1991–1993, Dr. Morasca was a faculty research assistant in the Institute for Advanced Computer Studies at the University of Maryland (UMIACS). He has authored several papers that have appeared at international conferences and in journals, including IEEE and ACM Transactions. He has also served on the Program Committee of a number of international conferences, and he is the guest co-editor of a special issue of *IJSEKE on Knowledge Discovery from Empirical Software Engineering Data*. Dr. Morasca's current research interests include: specification, verification, and validation of concurrent and real-time systems; formal methods; empirical studies in software engineering; and data mining. Dr. Morasca is a member of the IEEE Computer Society.



Victor R. Basili holds a BS degree from Fordham College; an MS degree from Syracuse University; and a PhD degree from the University of Texas at Austin. Dr. Basili is currently a professor of computer science at the University of Maryland. He is founder and a director in the Software Engineering Laboratory at NASA Goddard Space Flight, (established in 1976), and a recipient of the first Process Improvement Achievement Award by the IEEE, the SEI, and the NASA Group Achievement Award. Dr. Basili has authored over 130 journals and refereed conference papers; served as editor-in-chief of the IEEE *Transactions on Software Engineering*; been program chair and general chair of CSE'6 and ICSE'15; he is co-editor-in-chief of the *International Journal of Empirical Software Engineering*. He is a fellow of the IEEE, the ACM, and a member of the IEEE Computer Society.