Defining and Validating High-Level Design Metrics[1]

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

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

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

Abstract

The availability of significant metrics in the early phases of the software development process allows for a better management of the later phases, and a more effective quality assessment when software quality can still be easily affected by preventive or corrective actions. In this paper, we introduce and compare four strategies for defining high-level design metrics. They are based on different sets of assumptions (about the design process) related to a well defined experimental goal they help reach: identify error-prone software parts. In particular, we define ratio-scale metrics for cohesion and coupling that show interesting properties. An in-depth experimental validation, conducted on large scale projects demonstrates the usefulness of the metrics we define.

1 Introduction

Software metrics 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 [BR88, F91]. Most existing software metrics attempt to capture characteristics of software code [F91]; 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:

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

The production of better specifications and better designs reduces the need for extensive review, modification, and rewriting not only of code, but of specifications and designs as well. As a result, this allows the software organization to save time, cut production costs, and raise the final product's quality.

Early availability of metrics is a key factor to a successful management of software development, since it allows for

- 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;
- better software quality monitoring from the early phases of the life-cycle;
- quantitative comparison of techniques and empirical refinement of the processes to which they are applied;
- more accurate planning of resource allocation, based upon the predicted error-proneness of the system and its constituent parts.

In this paper, we will focus on high-level design metrics for software systems. A number of studies have been published on software design metrics in recent years. It has been shown that system architecture has an impact on maintainability and error-proneness [HK84, G86, R87, R90, S90, SB91, Z91, AE92, BTH93, BBH93]. These studies have attempted to capture the design characteristics affecting the ease of maintaining and debugging a software system. Most of the design metrics are based on information flow between subroutines or declaration counts. We think that, even though it provides an interesting insight into the program structure, this should not be the only strategy to be investigated, since many other types of program features and relationships are *a priori* worth studying. Moreover, there is a need for comparison between strategies in order to identify worthwhile research directions and build accurate prediction models.

Besides this focus on information flow, most of the existing approaches share two common characteristics. (1) They define metrics without making clear assumptions about the contexts (i.e., processes, problem domain, environmental factors, etc.) in which they can be applied (with the exception of [AE92], where this issue was partially addressed). This implies they should have general validity, and be applicable to different environments and problem domains. (2) There are not fully explicit goals, for whose achievement the metrics are defined. This may cause problems in their application, since they may be defined based on implicit assumptions which the context may not satisfy; interpretation, since their meaning is not clear; and validation [IS88, K88], since their relevance with respect to a clearly stated goal is not established.

The definition of universal metrics (like in physical sciences) is an acceptable long-term goal, which, however, is only achievable after we gain better insights into specific processes from specific perspectives in the short term. It is our opinion that the definition of a metric 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. In other words, the assumptions underlying the defined metrics should rely on a deep knowledge of the context and should be precisely related to a stated goal. After this, the defined metrics must undergo a thorough experimental validation, to assess their significance and usefulness with respect to the stated goals. Last, based on the experimental evidence, metrics may be refined and modified, to better achieve the goals and comply with the process characteristics.

The goal of the research documented in this paper is to define and validate a set of high-level design metrics to evaluate the quality of the high-level design of a software system with respect to its error-proneness, understand what high-level design characteristics are likely to make software error-prone, and predict the error-proneness of the code produced.

We introduce four families of metrics, which are based on different types of mathematical abstractions of program designs [MGBB90]. In particular, we introduce a family of metrics based on data declaration dependency links (Section 2.2.4). This strategy allows us to introduce metrics for cohesion (Section 2.2.4.1) and coupling (Section 2.2.4.2) [F91] that are characterized by interesting properties and are based on consistent principles. Such a consistency is important because it should facilitate future research on quantitative tradeoff mechanisms between coupling and cohesion, i.e., variations can be expressed using consistent measurement units. Other metric families include: metrics based on declaration counts (Section 2.2.1), metrics based on the USES relationships between modules [GJM92] (Section 2.2.2), and metrics based on the IS_COMPONENT_OF relationships [GJM92] (Section 2.2.3).

In addition, we experimentally compare and validate the metrics introduced in Section 2 on three NASA projects. The results are shown in Section 3. In Section 4, we summarize the lessons we have learned, and outline directions for future research activities based on these lessons.

2 Defining Metrics for High-level Design

In this section, we first introduce the basic concepts of high-level design and the terminology we will use in the paper (Section 2.1). We then define, based on the goals stated in Section 1 and context assumptions, four families of high-level design metrics (Section 2.2).

2.1 Basic Definitions

Our object of study is the high-level design of a software system. To define it, we will start from its elementary constituents: software modules.

In the literature, there are two commonly accepted definitions of modules. The first one sees a module as a routine, either procedural or functional, and has been used in most of the design measurement publications [M77, CY79, HK84, R87, S90]. 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 [BO87, GJM92]. In this view, a module is the implementation of an Abstract Data Type / Object. 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.

At a higher level of abstraction, modules 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 modules are IS_COMPONENT_OF [GJM92] relationships, and there is just one top level node, which is a module.

◊

In the remainder of this paper, we will define concepts and metrics 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 an LMH.

In the high-level design phase of a software system, only module and subroutine interfaces and their relationships are defined—module body and subroutine detail design 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 [GJM92] and IS_COMPONENT_OF relationships. No body information is yet available at this stage.

◊

2.2 Strategies to Define High-level Design Metrics

In this section, we investigate several strategies for defining high-level design metrics. This appears necessary at this stage of knowledge, where we can only rely on very limited theoretical and empirical ground to help us identify interesting concepts, relationships and objects of study. One of the results of this investigation is to provide directions to focus our research on a smaller set of strategies and concepts.

Some of the concepts introduced in this section cannot be directly mapped onto all imperative languages, because not all of them allow the implementation of Abstract Data Types/Objects. However, these concepts are shared by many modern programming languages.

As we said in the Introduction, context assumptions are necessary to define metrics that are applicable and useful. Therefore, we list a context assumption for each of the metrics of the four strategies we introduce below. We do not assume that all of these process assumptions are equally important, i.e., not all of the process characteristics we take into account have an equal impact on software error-proneness.

2.2.1 Declaration Counts

These metrics are counts of data declarations, associated with a software part, that are imported, exported or declared locally.

Metric 1: Local.
Local(sp) will denote the number of locally defined data declarations of a software part *sp*.

Assumption A-LO.
The count of declarations of a software part may be seen as a measure of size, which is known to be associated with errors, i.e, the larger the set of declarations, the more likely the errors.

◊

Metric 2: Global.
Global(sp) will denote the number of external data declarations visible from a software part *sp*.

Assumption A-GL.
The larger the number of external declarations visible in a software part, the larger the number of external concepts to be understood and used consistently, the higher the likelihood of error.

◊

Metric 3: Scope.
Scope(sp) will denote the number of external data declarations for which the data declarations of a software part *sp* are visible.

Assumption A-SC.
The larger the number of data declarations in the scope of the software part, the larger the number of contexts of use, the more likely it is to be inadequate to fulfill the needs of the declarations in the scope.

◊

2.2.2 Metrics Based on the USES Relationships

These metrics capture the dependencies between software parts based on the USES relationships of the system.

Metric 4: Imported Software Parts.
ISP(sp) will denote the number of software parts imported and used by a software part *sp*.

Assumption A-ISP.
The larger the number of used external software parts, the larger the context to be understood, the more likely the occurrence of an error.

◊

Metric 5: Exported Software Parts.
ESP(sp) will denote the number of software parts that use a software part *sp*.

Assumption A-ESP.
The larger the number of contexts of use of a software part, the larger the number of services it provides, the more flexible it must be, and, as a consequence, the more likely the occurrence of error.

◊

2.2.3 Metrics Based on the IS_COMPONENT_OF Relationships

These metrics capture information about the structure of the IS_COMPONENT_OF graph.

Metric 6: Maximum/Average Depth.
Max_Depth(sp) / Avg_Depth(sp) will denote the maximum/average depth of the nodes composing a software part *sp*.

Assumption A-M/A.
The larger the depth of a hierarchy, the larger the context information to be known in the lower nodes, the more likely the occurrence of error.

◊

Metric 7: Number of paths.
No_Paths(sp) will denote the number of complete paths (from root to leaf) within a a software part *sp*.

Assumption A-NOP.
The larger the number of paths, the larger the number of parent, sibling, and child relationships to be dealt with, the larger the complexity of the hierarchy, the higher the likelihood of error occurrence.

◊

2.2.4 Interaction-Based Metrics

In this section, we focus specifically on the dependencies that can propagate inconsistencies from data declarations to data declarations or subroutines when a new software part is integrated in a system. Those relationships will be called *interactions* and will be used to define metrics capturing cohesion and coupling within and between software parts, respectively. (Interactions linking subroutines to subroutines or data declarations will not be considered because they are, in the vast majority of cases, encapsulated in module or routine bodies and are therefore not detectable in our framework, which only takes into account high-level design.)

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 to 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, 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, which are useful to understand whether data declarations and subroutines are related to each other and therefore should be encapsulated into the same module (see Section 2.2.4.1 on cohesion).

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.

◊

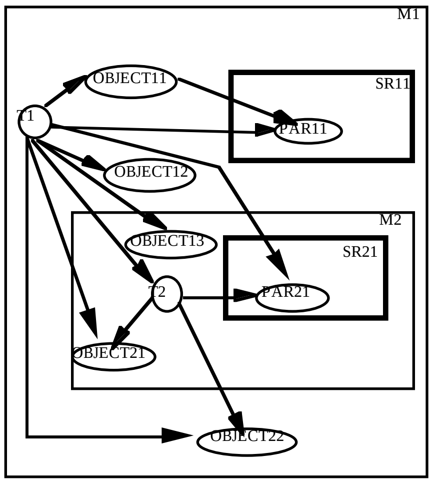
Whenever a data declaration DD-interacts with *at least one* of the data declarations contained in a subroutine interface, the DS-interaction relationship between the data declaration and the subroutine can be detected by examining the high-level design. For instance, from the Ada-like code fragment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

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

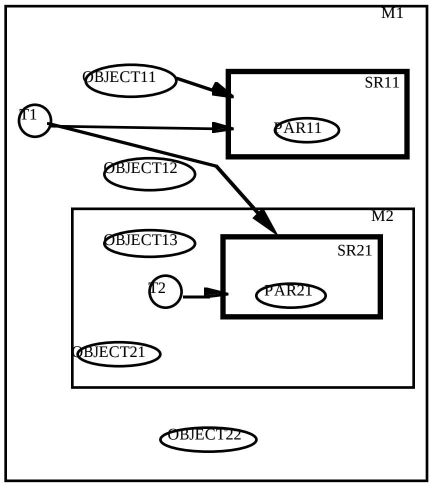
Figure 1. Program fragment

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 Figure 2, which shows DD- and DS-interaction graphs for the code fragment of Figure 1), data declarations are represented by rounded nodes, subroutines by thick lined boxes, modules by thin lined boxes, and interactions by arcs.

Next, we will define high-level design metrics for cohesion and coupling, based on the above definitions. It is generally acknowledged that system architecture should have low coupling and high cohesion [CY79]. 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 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.



(a)



(b)

Figure 2. DD-interaction (a) and DS-interaction (b) graphs for the program fragment in Figure 1

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., potential reuse) must be taken into account as well.

2.2.4.1 Cohesion

Cohesion captures the extent to which, in a software part, each group of data declarations and subroutines that are conceptually related belong to the same module. Based on

- an assumption (*A-CH*), which provides the rationale to define cohesion metrics;
- the concept of cohesive interactions, i.e., those interactions which contribute to cohesion;
- a set of properties (Properties 1-3) that cohesion metrics must have in order to measure cohesion

we now introduce a set of metrics (Metrics 8-11) to measure the degree of cohesion of a software part.

Assumption A-CH:

A high degree of cohesion is desirable because information related to declaration and subroutine dependencies should not be scattered across the system and 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 error-prone.

◊

Consistently 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 the data declarations and data declarations or subroutines declared in a module interface. We will capture this by means of *cohesive* interactions.

Definition 5: Cohesive Interaction.

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, with the exception of those DD-interactions between a data declaration and a subroutine formal parameter.

◊

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 routines seen as a whole. 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 data declarations and subroutines that are conceptually related belong to the same module. 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.

Remark.

It is worth reminding the reader that those relationships that cannot be 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. This issue will be further discussed later in this section.

◊

We base our cohesion metrics for software parts on cohesive interactions. Before defining them, we introduce the following three properties that they must satisfy in order to match our assumptions[1].

Property 1: Normalization.
Given a software part *sp*, the metric *cohesion(sp)* belongs to a specified interval *[0,Max]*, and *cohesion(sp)* = 0 if and only if *CI(sp)* is empty, and *cohesion(sp) = Max* if and only if *CI(sp)* includes all possible cohesive interactions.

◊

Normalization allows meaningful comparisons between the cohesions of different software parts, since they all belong to the same interval, and the extreme values of the cohesion range must represent the extreme cases. We will denote by *M(sp)* the maximal set of cohesive interactions of the software part *sp*, i.e., the set that includes all of *sp*'s possible cohesive interactions, obtained by linking every data declaration to every other data declaration and subroutine with which it can interact. Some care must be used in defining *M(sp)* for languages 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 a way that T1 interacts with T2 and T2 interacts with T1. We choose to count only one interaction between them. This is explained by the fact that a single interaction between two data declarations justifies their encapsulation in a single module/Abstract Data Type.

Property 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 $cohesion(sp_2) \geq cohesion(sp_1)$.

◊

Adding cohesive interactions to a a software part cannot decrease its cohesion.

Property 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 $cohesion(sp) \geq cohesion(sp')$.

◊

[1]Properties and metrics can be defined for module sets more general than software parts. However, for simplicity, we will provide them only for software parts.

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.

Based on the properties defined above, we introduce a cohesion metric for software parts.

Metric 8: Ratio of Cohesive Interactions (RCI) for a Software Part.
The Ratio of Cohesive Interactions for *sp* is

$$\mathrm{RCI}(sp)=\frac{|\mathrm{CI}(sp)|}{|\mathrm{M}(sp)|} \qquad (*)$$

It is straightforward to prove that *RCI(sp)* satisfies the above properties 1-3, and that, based on properties 1-3, it is defined on a ratio scale [F91]. Furthermore, *RCI(sp)* can also be computed as a weighted sum of the RCI(*m*)'s of the single modules *m* belonging to *sp*. From Formula (*), since cohesive interactions only occur within modules, but not across modules

$$|\mathrm{CI}(sp)| = \sum_{m\in sp} |\mathrm{CI}(m)|$$
$$|\mathrm{M}(sp)| = \sum_{n\in sp} |\mathrm{M}(n)|$$

so

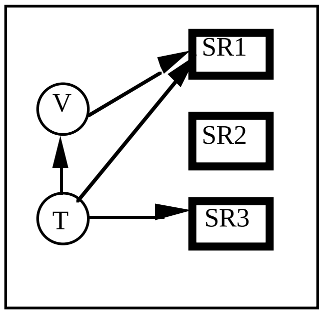
$$\mathrm{RCI}(sp)=\sum_{m\in sp}\frac{|\mathrm{CI}(m)|}{\sum_{n\in sp}|\mathrm{M}(n)|}$$

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

$$\mathrm{RCI}(sp)=\sum_{m\in sp}\frac{|\mathrm{M}(m)|}{\sum_{n\in sp}|\mathrm{M}(n)|}\;\frac{|\mathrm{CI}(m)|}{|\mathrm{M}(m)|}=\sum_{m\in sp}\frac{|\mathrm{M}(m)|}{\sum_{n\in sp}|\mathrm{M}(n)|}\,\mathrm{RCI}(m)$$

The weights represent the potential contribution of each module *m* belonging to the software part *sp* to the cohesion of the whole *sp*.

Figure 3 shows an example of cohesion computation for a single module. T denotes a type declaration, C a variable declaration, and SR1, SR2, and SR3 subroutine declarations.



RCI = 4/7 = 0.571

Figure 3. Cohesion example

Based on the above cohesion metric, we can define a threshold 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. 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.,

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

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

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

We want to emphasize, however, that, since cohesion is not the only characteristic relevant to software design, its increase should not be used as the *only* criterion on which to base such a decision.

The Role of Additional Information

Additional information to what is visible in the interfaces may be available at the end of high-level design. For instance, given the interface of a module *m*, the designers have at least a rough idea of which objects declared in *m* will be manipulated by a subroutine in *m*'s interface. It will be left to the person responsible for the metric program to decide whether or not it is worth collecting this kind of information, thus making the designer describe which objects will be accessed by which subroutines. Formatted comments may be a convenient way of conveying this information through module interfaces and therefore of automating the collection of this type of information.

For instance, from the code fragment in Figure 1, we cannot tell whether *OBJECT12* DS-interacts (as a global variable) with subroutine *SR11*. In this case, designers can answer in three different ways:

(1) *OBJECT12* DS-interact with *P11*
(2) *OBJECT12* does not DS-interact with *P11*
(3) the information they have is not sufficient

It is worth saying that answers of kind (2) provide valuable, though negative, information on the DS-interactions present in a system. For instance, in the code fragment on Figure 1, the designer may indicate the existence of a DD-interaction between object *OBJECT12* and *PAR11* and the lack of interaction between *OBJECT13* and *PAR21*. As a consequence, the computation of cohesion is affected. If we take into account this additional information, other alternative cohesion metrics 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 (they provide an answer similar to answer (1) above); 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 (they provide an answer similar to answer (1) above).

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 are not detectable from the high-level design and the designers explicitly exclude their existence.

Metric 9: Neutral Ratio of Cohesive Interactions (NRCI).
Unknown CIs are not taken into account

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

Metric 10: Pessimistic Ratio of Cohesive Interactions (PRCI).
Unknown CIs are considered as if they were known not to be actual interactions.

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

(This is equivalent to RCI(sp).)

Metric 11: Optimistic Ratio of Cohesive Interactions (ORCI).
Unknown CIs are considered as if they where known to be actual interactions

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

The above three metrics satisfy Properties 1-3, where *CI(sp)* is replaced by *K(sp)* ∪ *U(sp)*.

If PRCI(sp), NRCI(sp), and ORCI(sp) are all not undefined, 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.

Remark.
NRCI(sp) is undefined if and only if all interactions are unknown, i.e., no information is available on cohesive interactions. 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].

2.2.4.2 Coupling

In this section, we first give general definitions and assumptions on coupling, then, we present a set of metrics, and discuss the issue of genericity in the context of coupling. To address the particular issue of coupling, we will refer to the *import interactions* of a module *m* as all interactions going from a declaration outside *m* to a declaration inside *m*. Similarly, we define *export interactions* as going from a declaration located inside *m* to a declaration outside *m*.

Assumption A-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 error-prone.

◊

Definition 6: Import Coupling of a software part (IC).
Import Coupling is the extent to which a software part depends on imported external data declarations.

◊

Assumption A-EC:
Export coupling is related to how a software part is used in the system. The more often the software part is used, the larger the number of services it has to provide, the more flexible it needs to be, e.g., generic module. This may lead to errors.

◊

Definition 7: Export Coupling of a software part (EC).
Export coupling is the extent to which the data declarations of a software part affect the data declarations of the other software parts in the system.

◊

Import and export 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. We now provide properties that must be satisfied by both import and export coupling metrics.

Property 4: Non negativity
Given a software part *sp*, the metric $import\_coupling(sp) \geq 0$ (resp. $export\_coupling(sp) \geq 0$). $import\_coupling(sp) = 0$ (resp. $export\_coupling(sp) = 0$) if and only if *sp* does not have import (resp. export) interactions with other software parts.

◊

Property 5: Monotonicity
Let m_1 be a module and $II(m_1)$ (resp. $EI(m_1)$) its set of import (resp. export) interactions. If m_2 is a modified version of m_1 with the same sets of data and subroutine declarations and one more import (resp. export) interaction so that $II(m_2)$ (resp. $EI(m_2)$) includes $II(m_2)$ (resp. $EI(m_2)$), then $import\_coupling(m_2) \geq import\_coupling(m_1)$ (resp. $export\_coupling(m_2) \geq export\_coupling(m_1)$).

◊

Adding import (resp. export) interactions to a module cannot decrease its import (resp. export) coupling.

Property 6: Merging of Modules
The sum of the 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 or export interactions for the module resulting from merging the original modules.

It should be noted that, as opposed to cohesion, coupling is not a normalized metric. This comes from assumptions A-CH, A-IC, and A-EC (see Sections 2.2.4.1 and 2.2.4.2), where we state that cohesion is a *degree* of interdependence within a software part, whereas coupling is a *quantity* of dependencies between a software part and the rest of the system.

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

Metric 12: 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*.

Metric 13: Export Coupling
Given a software part *sp*, Export Coupling of *sp* (denoted by *EC(sp)*) is the number of DD-interactions between the data declarations within *sp* and the data declarations external to *sp*.

It is straightforward to prove that *IC(sp)* and *EC(sp)* satisfy the above properties 4-6, and that, based on properties 4-6, these metrics are defined on a ratio scale [F91].

Each box in Figure 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. Tij and OBJECTij data declarations represent respectively types and objects in module mi. FP3 represents a subroutine formal parameter. The IC and EC values for the modules in Figure 4 are computed as follows.

IC(m1) = 0	EC(m1) = 10
IC(m2) = 4	EC(m2) = 1
IC(m3) = 5	EC(m3) = 0
IC(m4) = 2	EC(m4) = 0

In the example of Figure 4, we see that m1 expectedly shows the largest export coupling.

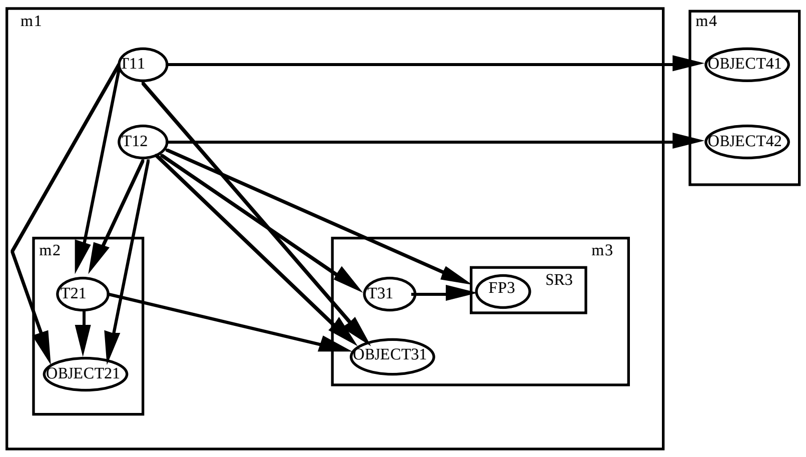


Figure 4. Calculation of IC and EC with non-generic modules only

Based on the definitions of *IC(sp)* and *EC(sp)*, we derive four related metrics, *DIC(sp)* (Direct Import Coupling), *TIC(sp)* (Transitive Import Coupling), *DEC(sp)* (Direct Export Coupling), *TEC(sp)* (Transitive Export Coupling). *DIC(sp)* and *DEC(sp)* only take into account *direct* interactions, whereas *TIC(sp)* and *TEC(sp)* only take into account *transitive* interactions. By their definitions, *IC(sp) = DIC(sp) + TIC(sp)*, and *EC(sp) = DEC(sp) + TEC(sp)*. This allows us to separately evaluate the impact of direct and transitive interactions on error-proneness, as we show in the experimental validation. In practice, the number of transitive interactions turns out to be much bigger than that of direct interactions, so *IC(sp) ≈ TIC(sp)* and *EC(sp) ≈ TEC(sp)*.

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 errors 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. When calculating export coupling, we take into account the DD-interactions between the data declarations of each of its instances and those of the software system. 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 Figure 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

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

IC(m1) = 2	EC(m1) = 4
IC(m2) = 3	EC(m2) = 0
IC(m3) = 4	EC(m3) = 0
IC(Gen_m) = 0	EC(Gen_m) = 5

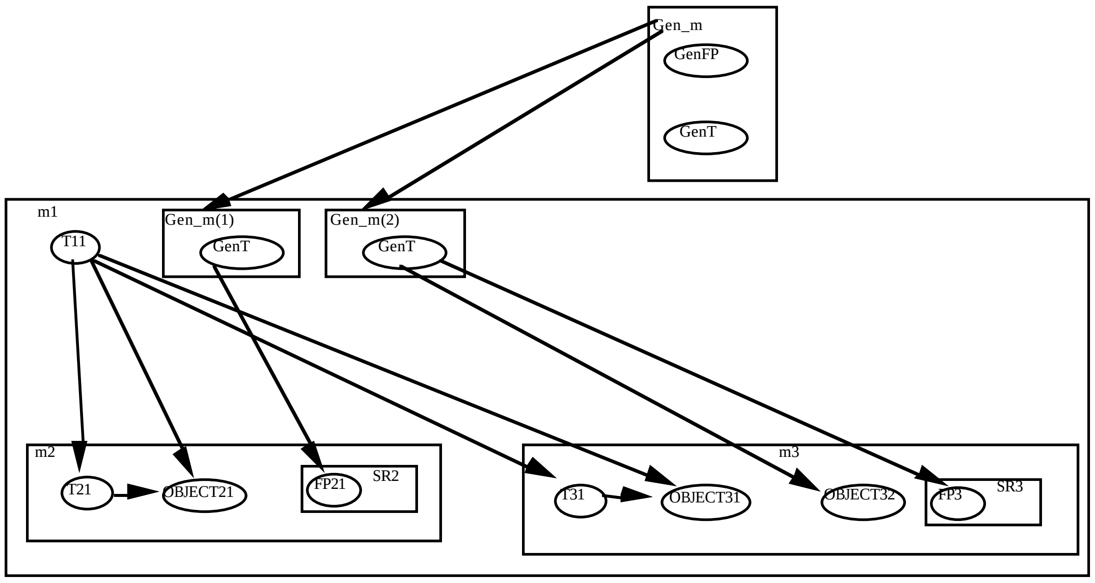


Figure 5. Generics when calculating coupling

3 Experimental Validation

The experimental validation has two main goals.

Goal 1

We want to find out which of the metrics defined above have a significant impact on the error-proneness of software parts. This allows us to

a. prove that high-level design information can be used to build significant indicators of software error-proneness
b. determine which of our assumptions about the development process (Section 2) are experimentally supported
c. compare the four strategies for defining high-level design metrics
d. identify the most promising research directions.

◊

Goal 2

We need to investigate dependencies between metrics, in order to determine which ones are complementary, and can be used in combination, and which ones capture similar phenomena.

◊

Section 3.1 presents the experimental design of the analysis, the project data sets used and the tool built to capture the discussed design metrics. Section 3.2 provides and discusses the results of a univariate analysis of the metrics. The significance of the metrics as predictors of error-prone software parts is assessed and the differences between systems are investigated. Section 3.3 investigates the results obtained when building multivariate classification models for detecting error-prone LMHs based on significant design metrics. The model results are assessed and the model functional structure is investigated.

3.1 Experiment Design

Experiment Layout

In order to validate software measurement assumptions experimentally, one can adopt two main strategies: (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 (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 metrics 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 degree of variation 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% of source code lines). The third system we studied (TONS) is an onboard navigation system for satellite that has been developed in the same environment and is about 180 Ada units and 50 Klocs large, with an extremely small rate of reuse (2% of source code lines). 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.

Tool

A tool analyzing the interface parts of Ada source code has been developed in order to capture the design characteristics of these systems. This tool is based on LEX&YACC [LY92] and extracts generic high-level design information about the visibility and interactions of the system declarations. This information is consequently used to compute the metrics presented in Section 2.2, and others that might be defined.

Analytical Model

The response variable we use to validate the design metrics is binary, i.e., Did an error ***not*** occur in an LMH? In order to analyze the impact of software metrics on the error-proneness of software parts, we used logistic regression, a classification technique [HL89] used in many experimental sciences, based on maximum likelihood estimation, and presented below. 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 [DG84][2]. In particular, we first used univariate logistic regression, to evaluate the impact of each of the metrics in isolation on error-proneness. Then, we performed multivariate logistic regression, to evaluate the relative impact of those metrics that had been assessed sufficiently significant in the univariate analysis (e.g., $\alpha < 0.20$ is a reasonable heuristic). This modeling process is further described in [HL89].

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):

$$\log\left(\frac{p}{1 - p}\right) = C_0 + C_1X_1 + C_2X_2 + \dots + C_nX_n$$

where p is the probability that no errors were found in a software part during the validation phase, and the X_i's are the design metrics included as predictors in the model (called *covariates* of the logistic regression equation). In the two extreme cases, i.e., when a variable is either non-significant or entirely differentiates error-prone software parts, the curve (between p and any single X_i, i.e., assuming that all other X_j's are constant) approximates a horizontal line and a vertical line respectively. In between, the curve takes a flexible S shape. However, since p is unknown, the coefficients C_i will be estimated through a likelihood function optimization. This procedure assumes that all observations are statistically independent. When building the regression equations, each observation was weighted according to the number of errors detected in each software part. The rationale is that each (non) detection of error is considered as an independent event. As a consequence, software parts where no errors were detected were weighted 1.

Goodness-of-fit for such a model is assessed via a statistic called R^2 (because similar in concept to the least-square regression coefficient of determination), belonging to the interval [0,1]. The higher R^2, the more accurate the model. However, as opposed to the R^2 of least-square regression, high R^2s are rare for logistic regression, for reasons whose explanation is well beyond the scope of this text. The interested reader may refer to [HL89] for a detailed introduction to logistic regression.

Tables 1 and 2 contain the results we obtained through, respectively, univariate and multivariate logistic regression on the three systems. We report those related to the metrics

[2]In addition, in order to confirm the obtained results, we used non-parametric tests for rank distributions such as the Mann-Whitney U test [CAP88]. Results appeared to be consistent across techniques and, in order to limit the amount of statistics provided to the reader and preserve the clarity of the text, we only show the results obtained with logistic regression.

that turned out to be the most significant ones across all three projects. For each metric, we provide the following statistics:

- C (appearing in both tables), the estimated regression coefficient. The larger the coefficient, the stronger the impact of the explanatory variable on the probability p.
- $\Delta\psi$ (appearing in Table 1 only), which is based on the notion of odd ratio [HL89], and provides an evaluation of the impact of the metric on the dependent variable. More specifically, the odd ratio $\psi(X)$ represents the ratio between the probability of not having an error and the probability of having an error when the value of the metric is X. As an example, if, for a given value X, $\psi(X)$ is 2, then it is twice more likely that the software part does not contain errors than that it does contain errors. The value of $\Delta\psi$ is computed by means of the following formula

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

  Therefore, $\Delta\psi$ represents the reduction/increase in the odd ratio when the value X increases by 1 unit. 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 one-tenth as the quantum for RCI increase with respect to which $\Delta\psi$ is computed.)
- α (appearing in both tables), the level of significance, which provides an insight into the accuracy of the coefficient estimates. The significance (α) of the logistic regression coefficients tells the reader about the probability for the coefficient to be different from zero by chance. Also, 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 [HL89] commonly used in the framework of logistic regression.

3.2 Univariate Analysis

Results

As Table 1 shows, all strategies presented in Section 2.2 provide significant metrics, but the strategy based on declaration counts. Therefore, these metrics, although based on simple and appealing concepts, do not appear to be significant predictors.

All the metrics based on exported declarations, i.e., *Local(sp)*, *ESP(sp)*, *EC(sp)*, *DEC(sp)*, and *TEC(sp)*, are not significant. Our explanation is that when an inconsistency exists between an exporting module *E* and an importing module *I*, *I* is more likely to be corrected, since *E* may export to other modules. Changing *E* is likely to require changing those other modules. Alternatively, a large amount of exports sometimes translates into a need for genericity but, for many declarations, just results into additional fields and dimensions. Therefore, the assumption underlying the export interactions metric appears somewhat questionable.

All the metrics based on the IS_COMPONENT_OF relation appear significant in the univariate analysis. However, they show a strong multicolinearity (i.e., the linear correlations are strong between metrics). Since *Avg_depth* is the best predictor in its category and in order to minimize the size of Table 1, only the *Avg_depth* results are shown.

A close analysis of the correlation matrix of the studied metrics shows that these results are not due to strong correlations between factors, e.g., when all factors are size predictors. Therefore, all the metrics in Table 1 seem to capture not only significant but different trends. This shows that most of the strategies are likely to be complementary and useful. This is confirmed by the multivariate results presented in Section 3.3.

	Projects	GOADA			GOESIM			TONS		
Strategy	Metrics	C	$\Delta\psi$	α	C	$\Delta\psi$	α	C	$\Delta\psi$	α
USES	ISP	-0.8	45%	0.000	-0.717	49%	0.002	-0.96	38%	0.000
I_C_O	Avg_Depth	-2.27	11%	0.000	-2.4	9%	0.000	-3.9	2%	0.000
Inter.	RCI	0.63	188%	0.000	0.215	124%	0.047	0.34	141%	0.001
Inter.	TIC	-0.016	98.5%	0.001	-0.017	98.3%	0.002	-0.02	98%	0.15
Inter.	DIC	-0.23	79%	0.000	-0.19	83%	0.001	-0.04	96%	0.19

Table 1. Univariate Analysis

Detailed Discussion

TIC and *DIC* do not appear to be significant in *TONS* (α = 0.19 and 0.15, respectively), whereas they are very significant in the two other systems. The analysis of the distribution of these factors in all three systems, respectively, shows that their standard deviation (σ) and median (m) are much smaller in *TONS*, i.e., with respect to *TIC*, $\sigma = 10$, $m = 2.5$ for *TONS* versus $\sigma = 32.74$, $m = 15.5$ for *GOADA*, $\sigma = 32.18$, $m = 59$ for *GOESIM*. As a consequence, any trend related to either *DIC* or *TIC* is very likely not to be visible in the *TONS* dataset. When considering that *TONS* is a significantly smaller system than the two other ones, results may be interpreted as follows: the distribution of import interactions is strongly dependent on the size of the system and input interaction metrics are likely to be mediocre predictors for small systems.

Comparing Models

From a more general perspective, variations across models (i.e., univariate regression equations) should be expected due to differences in project characteristics, i.e., size, application domain. However, 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. Because of the functional shape of logistic models, coefficients that may appear significantly different actually generate very similar models, e.g., In Table 1, coefficients -2.27 and -3.9 yield $\Delta\psi$'s of 11% and 2%, respectively. As a consequence, to evaluate the stability of the models, the reader should rather look at the $\Delta\psi$ column in Table 1. When doing so, only *RCI* appears to have a noticeable model instability even though the trends are consistent.

3.3 Multivariate Models

In this section, we present the results obtained by performing a stepwise multivariate logistic regression. Table 2 provides the estimated regression coefficients (C) and their significance (α) based on a *Wald test* [HL89], 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 0.2 (substituted by a *).

	Projects	GOADA		GOESIM		TONS	
Strategy	Metrics	C	α	C	α	C	α
USES	ISP	-0.9	0.04	*	*	-1.18	0.000
I_C_O	Avg_Depth	-1.8	0.003	-3.12	0.000	-5.62	0.000
Inter.	RCI	0.4	0.006	0.3	0.07	0.2	0.16
Inter.	TIC	-0.023	0.000	-0.02	0.005	*	*
Inter.	DIC	0.23	0.04	-0.13	0.04	-0.11	0.002

Table 2. Coefficients of Multivariate Models

Results

The very low levels of significance in Table 2 suggest that these metrics may be used in combination as indicators of error-prone LMHs. Indeed, when used in a multivariate model, many of these metrics are still significant and produce models that are more accurate than univariate models (Table 2). The best univariate R^2s are 0.115, 0.20 and 0.16 for *GOADA*, *GOESIM*, and *TONS*, respectively. In the same order, the multivariate R^2s are 0.21, 0.24, and 0.43. We can see that the results improved very significantly for *GOADA* and *TONS*.

Interaction-based metrics are more complex but worth collecting, since they are the only metrics defined at the declaration level that appeared significant. In addition, the average LMH depth was consistently selected as a very good indicator. This is likely to be an early measure of "size" of the LMH and is expectedly significant. Also, *ISP*, a metric similar to the notion of fan-in shows to be significant across projects (except in the multivariate *GOESIM* model for reasons explained below), while *ESP*, the equivalent measure for exports (based on the fan-out of LMHs) is not significant. As a consequence, a metric of the form $(\text{fan\_in} \cdot \text{fan\_out})^2$, suggested in numerous occasions in the literature [HK84, IS88, S90, Z91], does not appear to be significant. From a more general perspective, metrics based on imports, regardless of the associated concepts, appear to predict more accurately the error-proneness of software parts.

Comparing Models

Some variability in the estimated regression coefficients can be observed across projects in Table 2 and requires some discussion. In multivariate models, coefficients have a tendency to adjust, statistically, for other variables [HL89]. Sometimes, variables are weak predictors of the response variable when taken individually, and become more significant when integrated in a multivariate model. In Table 1, *DIC* showed, for *TONS*, a mediocre level of significance, whereas it appears to be a significant predictor in Table 2. Moreover, its coefficient is very unstable across projects and the trend is reversed (positive) for *GOADA* and *TONS*. When looking more carefully at the associations (not only the narrower concept of linear correlation) between metrics, it can be determined that this is the results of strong association between *DIC* and *ISP* in *GOADA* and *TONS*. These associations are a typical source of coefficient instability [DG84], e.g., the coefficient of *ISP* in *GOADA* varies from -0.9 to -0.39 when *DIC* is removed from the equation.

TIC remains non-significant because of its strong linear correlation ($R^2 = 0.76$) with *DIC* in the *TONS* dataset. Similarly, *ISP* does not appear significant in the *GOESIM* dataset because of a strong correlation with *DIC* ($R^2 = 0.50$). *RCI* in *TONS* shows a weaker significance ($\alpha = 0.16$) than in the univariate results and no strong linear correlation can be observed with the other metrics included in the multivariate equation. However, LMHs with large numbers of imported interactions are all located in the low part of the cohesion range. Such an *association* (likely to be spurious since it is not the case in the other datasets) with *DIC* is likely to affect the significance of *RCI* in a multivariate equation.

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.

4 Conclusion

This study has shown that statistical models of extremely good significance can be built based on *high-level* design information. In particular, we have determined accurate early predictors for error-prone software. Moreover, the results suggested that, at this stage of understanding, several strategies were worth investigating because none of them showed dominant trends, while most of them appeared to be complementary. In order to provide the practitioner with usable, well understood and validated models, software engineering researchers will have to keep refining and validating the existing metrics. There is still substantial room for improvement.

The stability of the impact of these metrics across projects allows us to draw optimistic conclusions about the use of such quality indicators. Using early quality indicators based on objective empirical evidence appears possible. However, it is very likely that this kind of indicators will behave differently across various domains of application and development environments.

Therefore, the use of such indicators should always be preceded by a careful empirical analysis of local error patterns and a thorough comparison across projects.

Our future work will be three-fold:

- Analyze more systems
- Validate further and refine the metrics we defined in this paper. The variations across environments and the study/comparison of different architectures is likely to give us interesting insights.
- Consistent with our current objectives, we will address the issues related to building metric based empirical models earlier in the life cycle. In particular, the next stage of this research will focus on defining and validating metrics for formal specifications.

Acknowledgments

We thank Giuseppe Calavaro and Chris Lott for their helpful comments on the earlier drafts of this paper.

REFERENCES

[AE92] W. Agresti and W. Evanco, "Projecting Software Defects from Analyzing Ada Designs," *IEEE Trans. Software Eng.*, 18 (11), November, 1992.

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

[BO87] G. Booch, "Software Engineering with Ada," Benjamin/Cumming Publishing Company, Inc., Menlo Park, California, 1987.

[BR88] V. Basili and H. Rombach,"The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, 14 (6), June, 1988.

[BTH93] L. Briand, W. Thomas and C. Hetmanski, "Modeling and Managing Risk early in Software Development," *International Conference on Software Engineering*, Maryland, May 1993

[CAP88] J. Capon, “Statistics for the Social Sciences”, Wadworth publishing company, 1988

[CY79] L. Constantine, E. Yourdon, "Structured Design," Prentice Hall, 1979

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

[DoD83] ANSI/MIL-STD-1815A-1983, Reference Manual of the Ada Programming Languages, U.S. Department of Defense, 1983

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

[G86] J. Gannon, E. Katz, V. Basili, "Metrics for Ada Packages: an Initial Study," Communications of the ACM, Vol. 29, N. 7, July 1986.

[GJM92] C. Ghezzi, M. Jazayeri, D. Mandrioli, "Fundamentals of Software Engineering," Prentice Hall, Englewood Cliffs, NJ, 1992

[HK84] S. Henry, D. Kafura, "The Evaluation of Systems' Structure Using Quantitative Metrics," Software Practice and Experience, 14 (6), June, 1984.

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

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

[LY92] J. Levine, T. Mason, D. Brown, "lex & yacc," O'Reilly & Associates, Inc., 1992

[M77] J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, 12(10):61-64, 1977

[MGBB90] A. Melton, D. Gustafson, J. Bieman, A. Baker, "A Mathematical Perspective for Software Measures Research," Software Engineering Journal, September 1990.

[R87] H. D. Rombach, "A Controlled Experiment on the Impact of Software Structure and Maintainability:," *IEEE Trans. Software Eng.*, 13 (5), May, 1987.

[R90] H. D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software,* March 1990.

[S90] M. Shepperd, "Design Metrics: An Empirical Analysis," *Software Engineering Journal*, January 1990.

[SB91] R. Selby and V. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.*, 17 (2), February, 1991.

[Z91] W. Zage, D. Zage, P. McDaniel, I. Khan, "Evaluating Design Metrics on Large-Scale Software," SERC-TR-106-P, September 1991.