A Unified Framework for Coupling Measurement in Object-Oriented Systems

Lionel C. Briand, John W. Daly, and Jürgen Wüst

Fraunhofer Institute for Experimental Software Engineering

Kaiserslautern, Germany.

ISERN-96-14

Abstract

The increasing importance being placed on software measurement has lead to an increased amount of research developing new software measures. Given the importance of object-oriented development techniques, one specific area where this has occurred is coupling measurement in object-oriented systems. However, despite a very interesting and rich body of work, there is little understanding of the motivation and empirical hypotheses behind many of these new measures. It is often difficult to determine how such measures relate to one another and for which application they can be used. As a consequence, it is very difficult for practitioners and researchers to obtain a clear picture of the state-of-the-art in order to select or define measures for object-oriented systems.

This situation is addressed and clarified through several different activities. First, a standardized terminology and formalism for expressing measures is provided which ensures that all measures using it are expressed in a fully consistent and operational manner. Second, to provide a structured synthesis, a review of the existing frameworks and measures for coupling measurement in object-oriented systems takes place. Third, a unified framework, based on the issues discovered in the review, is provided and all existing measures are then classified according to this framework. Finally, a review of the empirical validation work concerning existing coupling measures is provided.

This paper contributes to an increased understanding of the state-of-the-art: a mechanism is provided for comparing measures and their potential use, integrating existing measures which examine the same concepts in different ways, and facilitating more rigorous decision making regarding the definition of new measures and the selection of existing measures for a specific goal of measurement. In addition, our review of the state-of-the-art highlights several important issues: (i) many measures are not defined in a fully operational form, (ii) relatively few of them are based on explicit empirical models as recommended by measurement theory, and (iii) an even smaller number of measures have been empirically validated; thus, the usefulness of many measures has yet to be demonstrated.

Keywords: coupling, object-oriented, measurement.

1.0 Introduction

The market forces of today's software development industry have begun to place much more emphasis on software quality. This has lead to an increasingly large body of work being performed in the area of software measurement, particularly for evaluating and predicting the quality of software. In turn, this has lead to a large number of new measures being proposed for quality design principles such as coupling. High quality software design, among many other principles, should obey the principle of low coupling. Stevens *et al.*, who first introduced coupling in the context of structured development techniques, define coupling as "the measure of the strength of association established by a connection from one module to another" [SMC74]. Therefore the stronger the coupling between modules, i.e., the more inter-related they are, the more difficult these modules are to understand, change, and correct and thus the more complex the resulting software system. Some empirical evidence exists to support this theory for structured development techniques; see, e.g., [TZ81],[SB91].

The principle of low coupling has now been migrated to object-oriented design by Coad and Yourdon [CY91a, CY91b] and recent research has again lead to a large number of new coupling measures for object-oriented systems being defined. However, because coupling is a more complex software attribute in object-oriented systems (e.g., there are many different mechanisms that can constitute coupling) and there has been no attempt to provide a structured synthesis, our understanding of the state-of-the-art is poor. For example, because there is no standard terminology and formalism for expressing measures, many measures are not fully operationally defined, i.e., there is some ambiguity in their definitions. As a result, it is difficult to understand how different coupling measures relate to one another. Moreover, it is also unclear what the potential uses of many existing measures are and how these different measures might be used in a complementary manner. The fact that there also exists little empirical validation of existing object-oriented coupling measures means the usefulness of most measures is not supported.

To address and clarify our understanding of the state-of-the-art of coupling measurement in object-oriented systems requires a comprehensive framework based on a standard terminology and formalism. This framework can then be used (i) to facilitate comparison of existing measures, (ii) to facilitate the evaluation and validation of existing measures, and (iii) to support the definition of new measures and the selection of existing ones based on a particular goal of measurement.

The paper is organised as follows. Section 2.0 summarises the current state of coupling measurement in object-oriented system and provides detailed motivation for the need for the research performed in this paper. Section 3.0 introduces the notation and formalism required to conduct this research. Section 4.0 provides a comprehensive review and structured synthesis of existing object-oriented coupling frameworks and measures. The results of this review are then used to define a new unified framework for coupling measurement in object-oriented systems in Section 5.0. In Section 6.0, a review of empirical validation studies of coupling measures takes place. We conclude that our review and synthesis of the state of the art has shown that there are unexplored categories of coupling, i.e., for which there are no measures defined. In addition, there a few empirical studies investigating the validity of existing coupling measures. Those that do exist are weak in terms of the validation method used, the various threats to construct validity of the external attribute, and the underlying assumptions and theories are often either implicit or have not been considered.

2.0 Motivation

Object-oriented measurement has become an increasingly popular research area. This is substantiated by the fact that recently proposed in the literature are (i) several different frameworks for coupling and cohesion and (ii) a large number of different measures for object-oriented attributes such as coupling, cohesion, and inheritance. While this is to be welcomed, there are several negative aspects to the mainly ad hoc manner in which object-oriented measures are being developed. As neither a standard terminology or formalism exists, many measures are expressed in an ambiguous manner which limits their use. This also makes it difficult to understand how different measures relate to one another. For example, there are many different decisions that have to be made when defining a coupling measure - these decisions have to be made with respect to the goal of the measure and by defining an empirical model based on hypotheses. Unfortunately, many measures proposed in the literature do not have the motivation behind these decisions documented, making it difficult to understand the underlying assumptions of the measure, e.g., it is often unclear what constitutes a coupling connection between two classes or what the strength of the coupling connection measured is compared to types of connections used by other coupling measures. As a result, it is also unclear what the potential uses of existing measures are and how different coupling measures could be used in a complementary manner to obtain a more detailed picture of the coupling in an object-oriented system. In short, our understanding of existing coupling measures is not what it should be.

Several authors have tried to address this problem by introducing frameworks to characterise different approaches to coupling and the relative strengths of these, although, on their own, none of the frameworks could be considered comprehensive. There are three existing and quite different frameworks for object-oriented coupling (reviewed in detail in Section 4.2). First, Eder *et al.* identify three different types of relationships [EKS94]. These relationships, interaction relationships between methods, component relationships between classes, and inheritance between classes, are then used to derive different dimensions of coupling which are classified according to different strengths. Second, Hitz and Montazeri approach coupling by deriving two different types of coupling: object level coupling and class level coupling which are determined by the state of an object and the state of an object's implementation respectively [HM95]. Again different strengths of coupling are proposed. And third, Briand *et al.* constitute coupling as interactions between classes [BDM96]. The strength of the coupling is deter-

mined by the type of the interaction, the relationship between the classes, and the interaction's locus of impact. As none of the frameworks have been used to characterise existing measures to the different dimensions of coupling identified, the negative aspects highlighted above are still very prevalent ones. In our review of the literature, for example, we found more than thirty different measures[1] of object-oriented coupling. Consequently, it is not difficult to imagine how confusing the overall picture actually is.

To make a serious attempt to improve our understanding of object-oriented coupling measurement we have to integrate all existing frameworks into a unique theoretical framework, based on a homogenous and comprehensive formalism. A review has to be performed of existing measures and these measures have to be categorised according to the unified framework. This framework will then be a mechanism with which to compare measures and their potential use, integrate existing measures which examine the same concepts in a different manner, and allow more rigorous (and ease of) decision making regarding the definition of new measures and the selection of existing measures with respect to their utility. It should facilitate the evaluation and validation of coupling measures by ensuring that specific hypotheses are provided which link coupling measures to external quality attributes. It should also facilitate identification of dimensions of coupling which thus far have been neglected, i.e., for which there are no measures defined. Finally, the framework must be able to integrate new coupling measures as they are defined in the future. In that sense both the formalism and the framework must be extensible.

3.0 Terminology and Formalism

In the past, research within the area of software measurement has suffered from a lack of (i) standardised terminology and (ii) a formalism for defining measures in an unambiguous and fully operational manner (that is, a manner in which no additional interpretation is required on behalf of the user of the measure). As a consequence, development of consistent, understandable, and meaningful software quality predictors has been severely hampered. For example, Churcher and Shepperd [CS95a] and Hitz and Montazeri [HM96] have identified ambiguities in members of the well referenced object-oriented metrics suite by Chidamber and Kemerer [CK94]. To remedy this situation it is necessary to reach a consensus on the terminology, define a formalism for expressing software measures, and, most importantly, to use this terminology and formalism. Of course, the level of detail and scope of the terminology and formalism required are subject to the goal to be achieved.

To rigorously and thoroughly perform a review and a structured synthesis of software coupling measures we seek to define a terminology and formalism that is implementation independent and can be extended as necessary. This will allow all existing work to be expressed in a consistent, understandable, and meaningful manner and allow the measures reviewed to be expressed as operationally defined (additional interpretation of ambiguous measures is given when required). A disadvantage of this approach is that the reader must first be presented with the formalism before the review can begin in a meaningful fashion. Given the motivation for such an approach, however, it is argued that this is the only method to facilitate a rigorous and thorough review.

To prevent the reader having to read a complete terminology list we have provided a glossary in Appendix A, which includes definitions applicable to coupling in object-oriented systems and to measurement in general. This can be referenced as required. Where appropriate the terminology defined by Churcher and Shepperd [CS95b] has been used.

To express the coupling measures consistently and unambiguously the following formalism based on set and graph theory is presented. Note that for the sake of brevity we assume that the reader is familiar with common object-oriented principles and needs no explanation of them. For those readers not so familiar, simple explanations by means of examples are provided in Appendix B.

System

Definition 1: System, classes, inheritance relationships

An object-oriented system consists of a set of classes, *C*. There can exist inheritance relationships between classes such that for each class $c \in C$ let

- $Parents(c) \subset C$ be the set of parent classes of class c.

1. Note that this figure includes variations of the same measure, e.g., there are three versions of the RFC (response for a class) measure originally proposed by Chidamber and Kemerer [CK91].

- $Children(c) \subset C$ be the set of children classes of class c.
- $Ancestors(c) \subset C$ be the set of ancestor classes of class c.
- $Descendents(c) \subset C$ be the set of descendent classes of class c.

Methods

A class has a set of methods.

Definition 2: Methods of a class

For each class $c \in C$ let M(c) be the set of methods of class c.

A method can be either virtual or non-virtual and either inherited, overridden, or newly defined, all of which have implications for measuring coupling. It is therefore necessary to express the difference between these categories.

Definition 3: Declared and implemented methods

For each class $c \in C$, let

- $M_D(c) \subseteq M(c)$ be the set of methods *declared in c*, i.e., methods that *c* inherits but does not override or virtual methods of *c*
- $M_I(c) \subseteq M(c)$ be the set of methods *implemented in c*, i.e., methods that *c* inherits but overrides or non-virtual non-inherited methods of *c*

where $M(c) = M_D(c) \cup M_I(c)$ and $M_D(c) \cap M_I(c) = \varnothing$.

Definition 4: Inherited, overriding, and new methods

For each class $c \in C$ let

- $M_{INH}(c) \subseteq M(c)$ be the set of inherited methods of *c*.
- $M_{OVR}(c) \subseteq M(c)$ be the set of overriding methods of *c.*
- $M_{NEW}(c) \subseteq M(c)$ be the set of non-inherited, non-overriding methods of *c*.

For notational convenience, we also define the set of all methods in the system, *M(C)*.

Definition 5: *M(C)* the set of all methods

M(C) is the set of all methods in the system and is represented as $M(C) = \bigcup_{c \in C} M(c)$.

Methods have a set of parameters which, as they also influence coupling measurement, must be defined.

Definition 6: Parameters

For each method $m \in M(C)$ let *Par(m)* be the set of parameters of method *m.*

Method invocations

To measure coupling of a class, *c*, it is necessary to define the set of methods that $m \in M(c)$ invokes and the frequency of these invocations. Method invocations can be either static or dynamic; it is necessary to distinguish between these. Consequently, for each method $m \in M(C)$ the following sets are defined.

Definition 7: *SIM(m)* the set of statically invoked methods of *m*

Let $c \in C$, $m \in M_I(c)$ and $m' \in M(C)$. Then $m' \in SIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of *m* has a method invocation where *m´* is invoked for an object of static type class *d*.

Definition 8: *NSI(m, m´)* the number of static invocations of *m´* by *m*

Let $c \in C$, $m \in M_I(c)$ and $m' \in SIM(m)$. *NSI(m,m´)* is the number of method invocations in *m* where *m´* is invoked for an object of static type class *d* and $m' \in M(d)$.

Definition 9: *PIM(m)* the set of polymorphically invoked methods of *m*

Let $c \in C$, $m \in M_I(c)$ and $m' \in M(C)$. Then $m' \in PIM(m) \Leftrightarrow \exists d \in C$ such that $m' \in M(d)$ and the body of *m*

has a method invocation where *m´* may, because of polymorphism, be invoked for an object of dynamic type *d*.

Definition 10: *NPI(m, m´)* the number of polymorphic invocations of *m´* by *m*

Let $c \in C$, $m \in M_I(c)$ and $m' \in PIM(m)$. *NPI(m,m´)* is the number of method invocations in *m* where *m´* can be invoked for an object of dynamic type class *d* and $m' \in M(d)$.

As a result of polymorphism, one method invocation can contribute to the *NPI* count of several methods.

Attributes

Classes have attributes which are either inherited or newly defined. Attributes are modelled using a similar formalism to that of methods.

Definition 11: Declared and implemented attributes

For each class $c \in C$ let A(c) be the set of attributes of class *c*. $A(c) = A_D(c) \cup A_I(c)$ where

- $A_D(c)$ is the set of attributes declared in class *c* (i.e., inherited attributes).
- $A_I(c)$ is the set of attributes implemented in class *c* (i.e., non-inherited attributes).

Again, for notational convenience, we define the set of all attributes in the system, A(C).

Definition 12: *A(C)* the set of all attributes

A(C) is the set of all attributes in the system and is represented as $A(C) = \bigcup_{c \in C} A(c)$.

Attribute references

Methods may reference attributes. It is sufficient to consider the static type of the object for which an attribute is referenced because attribute references are not determined dynamically. For the discussion of measures later, it must be possible to express for a method, *m*, the set of attributes referenced by the method:

Definition 13: *AR(m)*

For each $m \in M(C)$ let *AR(m)* be the set of attributes referenced by method *m*.

Types

Attributes and parameters have types which all can contribute to coupling. The programming language provides a basic set of built-in types; the user can define new class types as well as traditional types (e.g., records, enumerations).

Definition 14: Basic types and user-defined types

- *BT* is the set of built-in types provided by the programming language (e.g., integer, real, character, string).
- *UDT* is the set of user-defined types of global scope (e.g., records, enumerations).

The type of an attribute or parameter either is a class, a built-in type or a user-defined type. Thus, the set ***T*** of available types in the system is defined as follows:

Definition 15: ***T*** the set of available types

The set ***T*** of available types in the system is $\boldsymbol{T} = BT \cup UDT \cup C$.

The next definition determines how the type of attributes and parameters will be denoted.

Definition 16: Types of attributes and parameters

For each attribute $a \in A(C)$ the type of attribute *a* is denoted by $T(a) \in \boldsymbol{T}$.

For each method $m \in M(C)$ and each parameter $v \in Par(m)$ the type of parameter *v* is denoted by $T(v) \in \boldsymbol{T}$.

No distinction is made between pointers, references, or arrays and the type they are derived from.

Predicates

To ensure consistency the predicate uses must be defined.

Definition 17: uses

$$uses(c, d) \Leftrightarrow (\exists m \in M_I(c) : \exists m' \in M_I(d) : m' \in PIM(m)) \vee (\exists m \in M_I(c) \exists a \in A_I(d) : a \in AR(m))$$

A class *c* uses a class *d* if a method implemented in class *c* references a method or an attribute implemented in class *d*.

C++ specific extensions

In C++, a class *c* can declare a class *d* its friend, which means that *d* is thus granted access to non-public elements of *c*. We must be able to specify for a class *c*, which are the friends of class *c*, and which classes declare class *c* their friend, as this is likely to have an influence on the strength of coupling between the classes.

Definition 18: Friends

For each class $c \in C$, we define the set $Friends(c) \subset C$ of friend classes of *c*.

For each class $c \in C$, the set of inverse friends of *c* (i.e., the set of classes that declare *c* their friend) is defined as $Friends^{-1}(c) = \{d | d \in C \wedge c \in Friends(d)\}$.

In hybrid languages such as C++, which also provide features found in the procedural paradigm, it is possible to take a pointer of a method, and pass this pointer to another method. The formalism must be able to express this.

Definition 19: Passing of pointers to methods

For methods $m, m' \in M(C)$, we define $PP(m, m')$ to be the number of invocations of *m*, where a pointer to m' is passed to *m* (via parameter). Such method invocations can be located in the body of any method in the system, not only in methods of the classes where *m* and m' are defined.

The notation and formalism defined, a mechanism is now available to express existing coupling frameworks and measures in a consistent and precise manner.

4.0 Survey of Coupling Measurement Frameworks and Measures

In this section we perform a comprehensive survey and critical review of existing frameworks and measures for coupling in object-oriented systems. The section is organized as follows. In Section 4.1, we present existing frameworks for coupling. These frameworks are then discussed and compared in Section 4.2. In Section 4.3, a survey and review (which includes theoretical validation) of existing coupling measures takes place.

4.1 Existing frameworks for coupling

Frameworks for coupling in object-oriented systems have been proposed by Eder *et al.* [EKS94], Hitz and Montazeri [HM95], and Briand *et al.* [BDM96]. In each framework different types of class, method, and object coupling are identified.

4.1.1 Framework by Eder *et al.* [EKS94]

Eder *et al.* use the definition of coupling provided by Stevens *et al.* [SMC74]. The authors identify the following types of relationships:

- Interaction relationships between methods. This type of relationship is caused by message passing.
- Component relationships. Each object has a unique identifier (the object identity). An object *o* may reference another object *p* using the identifier of object *p*. This introduces a component relationship between the classes of *o* and *p*.
- Inheritance relationships between classes.

From these relationships three dimensions of coupling are derived.

1. *Interaction coupling*: two methods are interaction coupled if

   (a) one method invokes the other, or
   (b) they communicate via sharing of data.

   Interaction coupling between method *m* implemented in class *c* and method *m´* implemented in class *c´* contributes to interaction coupling between the classes *c* and *c´*.

2. *Component coupling*: two classes *c* and *c´* are component coupled, if *c´* is the type of either

   (a) an attribute of *c*, or
   (b) an input or output parameter of a method of *c*, or
   (c) a local variable of a method of *c*, or
   (d) an input or output parameter of a method invoked within a method of *c.*

3. *Inheritance coupling*: two classes *c* and *c´* are inheritance coupled, if one class is an ancestor of the other.

For each dimension of coupling, the authors identify different strengths of coupling (listed below from strongest to weakest).

Interaction coupling: The definition of interaction coupling is most similar to the original definition of coupling. Therefore, Eder *et al.* use the types of coupling proposed by Myers [Mye78] and adapt these to object-oriented systems (see Appendix D for a short summary of Myers' types of coupling).

- *Content*: Method accesses implementation of another. Implementation here means the non-public part of the class interface. In C++, for instance, a method *m* may be declared "friend" of a class *c*. Method *m* can then invoke private methods of class *c*. Access to implementation constitutes a breach of the information hiding principle and is considered the worst type of coupling.
- *Common*: Methods communicate via unstructured, global, shared data space. The authors cannot give an example for an unstructured global shared data space, because there are no object-oriented languages which support them. Apparently, this type of coupling is only listed in order to be consistent with Myers' categories.
- *External*: Methods communicate via structured, global, shared data space (e.g., a public attribute of a class). Eder *et al.* find that this is a violation of the "locality principle of good software design", without specifying any further what they mean by that.
- *Control*: Methods communicate via parameter passing only, the called method controls the internal logic of the calling method. For instance, the called method may determine the future execution of the calling method. A change of the implementation of the called method will most likely affect the calling method (change dependencies).
- *Stamp*: Methods communicate via parameter passing only, the called method does not need all of the data it receives. This constitutes an avoidable dependency between methods. E.g., if the data structure of a parameter of a method is changed, possible effects of this change on the method have to be considered. If the parameter is unused, a change of the parameter's data structure will not have any effects. The effort spent to discover that the change has no effects can be saved by avoiding stamp coupling.
- *Data*: Methods communicate via parameter passing only, the called method needs all the data it receives. This is the best type of coupling (besides no coupling at all), because it minimizes the change dependencies.
- *No direct coupling*: No direct interaction coupling between two methods occurs.

Eder *et al.* first consider only direct interaction coupling between two methods. Their definition is then expanded to indirect interaction coupling via transitive method invocations.

Component coupling: There are four degrees of component coupling between classes (listed below from strongest to weakest).

- *Hidden*: Component coupling does not manifest itself in code. For instance, if a class *c* contains a cascaded method invocation such as *a.m1().m2()*, the type of the object returned by *m2* need not be explicit in the interface or body of class *c*. It can be found in the interface of the class of the object returned by method *m1*. That is, in order to detect occurrences of hidden coupling where class *c* is involved, we also have to look at the interfaces of other classes.
- *Scattered*: Component coupling manifests itself in the body of the class only (cases (c) and (d) in the above definition). Consequently, the body of the class has to be searched in order to detect occurrences of this type of coupling.

- *Specified*: Component coupling manifests itself in the interface (cases (a) and (b)). It is sufficient to search the interface of the class for occurrences of this type of coupling.
- *Nil*: No component coupling.

Inheritance coupling: There are four degrees of inheritance coupling (listed below from strongest to weakest).

- *Modification*: Inheriting class changes at least one inherited method in a manner that violates some predefined "good practice" rules. Eder *et al.* provide examples of such rules as "the signature of an inherited method *m* may only be changed by replacing the type of a parameter of *m*, say class *d*, with a descendent of class *d*", "an inherited method must not be deleted from the class interface", and "if a method is overridden, the overriding method must keep the same semantics as the overridden method". The predefined rules applied will, to a certain extent, depend on the used design methodology and programming language, but it should be obvious that such rules are subjective and not easily measured automatically. Modification coupling is the strongest type of inheritance coupling because information inherited from the parent class is modified or deleted in a manner which cannot be justified in the context of inheritance. Two types of modification coupling exist.
    1. *Signature modification:* not only the implementation of at least one inherited method is changed, but the signature of the method is also changed.
    2. *Implementation modification:* the implementation of at least one inherited method is changed. This degree of coupling is weaker than the previous type because the signature of the method is not changed.
- *Refinement*: Inheriting class changes at least one inherited method but the change is made adhering to the predefined "good practice" rules. Refinement coupling is weaker than modification coupling because the inherited information is changed only according to the predefined rules. However, problems can still occur as a result of refinement coupling, e.g., changes to the signature of an inherited method will restrict the use of polymorphism even if the intended semantics of the method are not changed. Again, like modification coupling, there exist two different types of refinement coupling.
    1. *Signature refinement:* not only the implementation of at least one inherited method is changed, but the signature of the method is also changed.
    2. *Implementation refinement:* the implementation of at least one inherited method is changed. This degree of coupling is weaker than the previous type because the signature of the method is not changed.
- *Extension*: Inheriting class changes neither the signature nor the body of any inherited method; only new methods and attributes are added.
- *Nil*: No inheritance relationship between two classes.

4.1.2 Framework by Hitz and Montazeri [HM95]

Hitz and Montazeri approach coupling by defining the state of an object (the value of its attributes at a given moment at run-time), and the state of an object's implementation (class interface and body at a given time in the development cycle). From these definitions, they derive two "levels" of coupling:

- *Class level coupling* (CLC): CLC represents the coupling resulting from implementation dependencies between two classes in a system during the development lifecycle.
- *Object level coupling* (OLC): OLC represents the coupling resulting from state dependencies between two objects during the run-time of a system.

According to Hitz and Montazeri, CLC is important when considering maintenance and change dependencies because changes in a server class may lead to changes in client classes. The authors also state that OLC is relevant for run-time oriented activities such as testing and debugging. For each of these levels of coupling, the authors identify a series of factors determining the strength of coupling.

1. Class level coupling. CLC can occur if a method of a class invokes a method or references an attribute of another class. In the following, let *cc* be the accessing class (client class), *sc* be the accessed class (server class). The factors determining the strength of CLC between *cc* and *sc* are:

- *Stability of sc*:

- *sc* is stable: interface or body of *sc* is unlikely to be changed (for instance due to changing requirements). Typically, basic types provided by the programming language, or classes imported from standard libraries are stable.
- *sc* is unstable: a class depending on an unstable server class is considered worse that depending on a stable class because a change to *sc* means potential change to *cc*. Typically, problem domain classes are unstable. Two cases must be considered.
  1. only the body of *sc* is likely to be changed.
  2. the interface of *sc* may also be modified. This case is considered the more harmful modification.

- *Type of access*:
  - "Access to interface": *cc* invokes a method of *sc*.
  - "Access to implementation": *cc* references an attribute of *sc*.

  Access to implementation is considered stronger coupling as it constitutes a violation of the information hiding principle.
- *Scope of access*: Determines where *sc* is visible within the definition of *cc*. Within this scope, a change to *sc* may have an impact on *cc*. The larger the scope, the stronger the classes are coupled. The authors identify five cases which can be separated into two categories: (i) a reference to *sc* may occur in any method of *cc* and (ii) a reference to *sc* can occur only through a particular method of *cc* (this becomes clear below). Category (i) comprises of three cases:
  - *sc* is the type of an attribute of *cc*.
  - *sc* is an ancestor of *cc*.
  - *sc* is the type of a global variable.

  Category (ii) comprises of two cases:
  - *sc* is the type of a local variable of a method of *cc.*
  - *sc* is the type of a parameter of a method of *cc*.

2. Object level coupling. For the discussion of OLC (object level coupling), let o_{sc} be an object of type *sc*, o_{cc} an object of type *cc*. Three factors influence the strength of coupling between objects o_{sc} and o_{cc}:

- *Type of access*: o_{cc} accesses interface of o_{sc} or o_{cc} accesses implementation of o_{sc} (same distinction and implications for strengths of coupling as for CLC).
- *Scope of access*: The smaller the scope of access the weaker the coupling between the objects. For o_{cc} to be able to access o_{sc} and contribute to OLC object o_{sc} must be either (listed in increasing size of scope)
  1. a parameter of a method of o_{cc}
  2. a "non-native" part of o_{cc}, that is, o_{sc} is not an object inherited from a superclass of o_{cc} nor is it encapsulated (aggregation) within o_{cc} nor is it a local variable to one of o_{cc} methods
  3. a global object.
- *Complexity of interface*: In the case that o_{cc} sends a message to o_{sc}, the number of parameters of the invoked method should be considered. The more parameters passed, the stronger the coupling between objects.

4.1.3 Framework by Briand *et al.* [BDM96]

An earlier approach ([BMB93], [BMB94]) to measure coupling in object-based systems such as those implemented in Ada is adapted to C++ by expanding it to include inheritance and friendship relations between classes. In contrast to the two previous frameworks, this framework focuses solely on coupling relationships available during the high level design phase. The motivation behind this decision is that eliminating design flaws and errors early before they can propagate to subsequent phases can save substantial amounts of money. As a result of the decision to focus on early design information, this framework concentrates on coupling as caused by interactions that occur between classes. Three different facets are identified that determine the kind of interaction:

1. *Type of interaction*: Determines the mechanism by which two classes are coupled. Different types of interaction are specified to determine if a particular type more accurately indicates fault likelihood.

- *Class-attribute interaction*: There is a class-attribute interaction between classes *c* and *d*, if class *c* is the type of an attribute of class *d* (i.e., if aggregation occurs).

- *Class-method interaction*: Let m_d be a method of class *d*. There is a class-method interaction between classes *c* and *d*, if
  - class *c* is the type of a parameter of method m_d
  - class *c* is the return type of method m_d.
- *Method-method interaction*: Let m_c be a method of class *c*, m_d be a method of class *d*. There is a method-method interaction between classes *c* and *d*, if
  - m_d directly invokes m_c
  - m_d receives via parameter a pointer to m_c thereby invoking m_c indirectly.

2. *Relationship*: In C++, two classes *c* and *d* can have one of three basic relationships:

- *Inheritance*: class *c* is an ancestor of class *d* or vice versa. This category is specified because the use of inheritance apparently contradicts the notion of minimizing coupling and should be considered separately. Coad and Yourdon proposed a design principle which recommends high coupling between a class and its parents, and low coupling between classes not related via inheritance [CY91a], [CY91b].
- *Friendship*: class *c* declares class *d* its friend which grants class *d* access to non-public elements of *c*. This category is specified because it breaks encapsulation and thus violates the information-hiding principle.
- *Other*: no inheritance or friendship relationship between classes *c* and *d*.

3. *Locus*: the "locus of impact" of an interaction. If class *c* is involved in an interaction with another class, a distinction is made between

- *export*: class *c* is the server class in the interaction, and
- *import*: class *c* is the client class in the interaction.

The motivation for this distinction is to investigate whether direction is important for predicting the fault-proneness of a class.

In the definition of this framework, Briand *et al.* deliberately assign no strengths to the different kinds of interactions they propose. The authors state such strengths should be derived from empirical validation which can then be used to define measures on an interval or ratio scale. Briand *et al.* pose several hypotheses regarding these facets of coupling and investigate these empirically with respect to prediction of fault prone classes (see Section 6.0 for discussion).

4.2 Discussion and comparison of frameworks

A precise comparison of the frameworks shows there are differences in the manner in which coupling is addressed. One reason for this is the different objectives of the frameworks. For example, Briand *et al.* examined only early design information to investigate potential early quality indicators while the other authors investigated information mainly available at low level design and implementation; hence differences are found in the mechanisms that constitute coupling. A second reason is that some of the issues dealt with by one set of authors are considered to be subjective and too difficult to measure automatically. For example, the stability of an individual class (addressed by Hitz and Montazeri) is not something which can be easily determined unless, say, all problem domain classes are classified as unstable. The following subsections discuss in detail the significant differences between the frameworks and what can be learned from these differences.

4.2.1 The mechanisms that constitute coupling

In Table 1, the mechanisms that constitute coupling according to each of the frameworks are presented. Each row represents one mechanism, an "X" indicates that the mechanism is covered by the framework in the respective column. The mechanisms are numbered for reference purposes.

There is little overlapping of the frameworks: method invocation is the only mechanism common to all three frameworks. Mechanisms 5 and 6 are common to the frameworks by Briand *et al*. and Eder *et al.* All other mechanisms are unique to one of the frameworks.

Mechanism 9 (inheritance) is of a different nature than the other mechanisms. If two classes are connected via one of the mechanisms 1 to 8, then there is an actual client-server relationship between these classes: one class

#	Mechanism	Eder *et al.* [EKS94]	Hitz & Montazeri [HM95]	Briand *et al.* [BDM96]
1	methods share data (public attributes etc.)	X		
2	method references attribute		X	
3	method invokes method	X	X	X
4	method receives pointer to method			X
5	class is type of a class' attribute (aggregation)	X		X
6	class is type of a method's parameter or return type	X		X
7	class is type of a method's local variable	X		
8	class is type of a parameter of a method invoked from within another method	X		
9	class is ancestor of another class	X		

Table 1: Comparison of mechanisms that constitute coupling

uses the other. If two classes are connected via mechanism 9, i.e., one class is the ancestor of the other, then there *can* (and probably should), but *need not* be any client-server relationship between the classes. The client-server relationship and the inter-dependencies this entails do not necessarily exist.

The coupling mechanisms differ in the development phase in which they become applicable. For instance, attribute references and method invocations (mechanisms 2 and 3) are completely known only after implementation. In contrast, aggregation is visible in the class interface and is typically available before implementation starts. Coupling mechanisms that are applicable early in the development process are particularly interesting. If, for instance, they help in identifying fault-prone classes, this information could be used to select classes which are to undergo formal verification, to allocate the best people to the most fault-prone parts of the design, or to select the optimal design from a series of design alternatives before these are implemented. However, the later the development phase, the more detailed the description of the system under development, the more detailed the analysis of coupling.

4.2.2 Strength of coupling

The strength of coupling between two classes is determined by two aspects:

- the frequency of connections between the classes, and
- the types of connections between the classes.

The first aspect, how to count the frequency of connections between classes, has to be ultimately resolved when defining the measures. This aspect has not been addressed by any framework probably because the information required to make this decision is available only after the source code is developed. In Section 4.3, where the definitions of various proposed coupling measures are compared, it is shown that there are a number of different ways to count the frequency of connections between classes.

Different types of coupling have different strengths. Eder *et al.* and Hitz and Montazeri assign strengths to the types of coupling they identified by defining a partial order on the set of coupling types used in their frameworks. That is, for any two types of connections within each framework, they define if one is stronger than the other, if both have equal strength, or if their strengths are not comparable. It is important to note that the definition of such a partial order is to some degree subjective and requires empirical validation. Furthermore, the validity of a given order will clearly depend on the concrete measurement goal, i.e., different measurement goals can require different (partial) orders. For instance, assuming regression testing by means of structural testing based on control flow analysis is performed, we wish to estimate the effort to test a class based on the amount of import coupling of the class. We would be interested in method invocations because that influences the flow of control. We would not but be interested in references to attributes because these have no impact on the flow of control. If, on the

other hand, we want to characterize the understandability of the class based on import coupling, direct references to attributes are likely to be equally important as method invocations.

Briand *et al.* define a set of measures which count for each interaction type of their framework the number of interactions a class has with other classes. Empirical validation is then conducted to evaluate their potential of identifying fault-prone classes (see Section 6.0 for discussion). That is, strengths are empirically assigned to the different types of coupling.

4.2.3 Direction of coupling

The framework by Briand *et al.* explicitly distinguishes import and export coupling. Consider two classes *c* and *d* being coupled through one of the mechanisms mentioned above. This introduces a client-server-relationship between the classes: the client class uses (imports services), the server class is being used (exports services). This distinction is important. A class which mainly imports services may be difficult to reuse in another context because it depends on many other classes. On the other hand, defects in a class which mainly exports services are particularly critical as they may propagate more easily to other parts of the system and are more difficult to isolate. We conclude that the direction of coupling measured directly influences the possible goals of measurement.

If two methods are coupled through “common” or “external” coupling according to the framework by Eder *et al.*, we cannot make a distinction between client and server so both methods would be clients. Note that with pure object-oriented languages this would not occur. If the global data space is a variable whose type is a class, this class could be considered the server.

4.2.4 Direct and indirect coupling

Eder *et al.* derive “indirect interaction relationships between methods” from “direct interaction relationships” using the transitive closure of direct interaction relationships. This idea can be applied to all kinds of coupling. If a class c_1 uses a class c_2, which in turn uses a class c_3, class c_1 is indirectly coupled to c_3: a defect or modification in class c_3 may not only affect the directly coupled class c_2, but also the indirectly coupled class c_1. As an extreme case, consider a circular chain of coupled classes (class c_i uses class c_{i+1} for $i=1,2,...,n-1$, and class c_n uses c_1). Each class is directly coupled with two other classes (import and export coupling). However, each class in the chain indirectly uses and is being used by every other class.

Briand *et al.* based their framework on the work described in [BMB94]. In [BMB94], high-level design measures for coupling and cohesion in object-based systems were defined and validated with respect to their potential of identifying fault-prone modules. The coupling measures included measures for direct and indirect coupling. The measures for direct coupling were found to be useful predictors, those for indirect coupling, however, not. Therefore, in [BDM96] Briand *et al.* did not include the distinction between direct and indirect coupling in their framework (because the framework has primarily been defined to derive coupling measures for the identification of fault-prone classes).

4.2.5 Stability of server class

This point is unique to the framework by Hitz and Montazeri. Using a stable class is better than using an unstable class, because modifications which could ripple through the system are less likely to occur. “Stability of the server class” could, for instance, be used to distinguish between classes imported from standard libraries (which usually are not being modified and thus are stable), and problem domain classes (which are unstable). Note that stability of a server class is a subjective concept which is difficult to measure automatically in a manner other than that suggested above.

4.2.6 Inheritance

An aspect unique to object-oriented systems is how inheritance influences coupling. This aspect is addressed in some detail by Eder *et al.* and Briand *et al.* For the discussion, let us consider the case where a method of one class invokes a method of another. The question is: since both the invoking and the invoked method can be either declared or implemented in their classes, how does this affect coupling between the classes?

```
class c1 {
     d1 *o1;
   public:
     void mc1();
     void mc2();
};
void c1::mc1() { o1->md1(); }
void c1::mc2() { o1->md2(); }


class c2 : public c1 {
     d2 *o2;
   public:
     void mc2(); /* redefined */
     void mc3();
};
void c2::mc2() { o2->md2(); }
void c2::mc3() {
       mc1(); o2->md1();
       o2->md2();
}
```

```
class d1 {
   public:
     virtual void md1();
     virtual void md2();
};
void d1::md1() { ... }
void d1::md2() { ... }


class d2 : public d1 {
   public:
     void md2(); /* override */
     void md3();
};
void d2::md2() { ... }
void d2::md3() { ... }


class d3 : public d1 {
   public:
     void md3();
   };
void d3::md3() { ... }
```

FIGURE 1. Example inheritance hierarchy

Consider the example in Figure 1. There are four points worth noting here:

a) If a method invokes another method, the invoking method contributes to import coupling of its class from other classes. What if the invoking method is inherited? In Figure 1, method *mc1* in class *c1* invokes *md1* in *d1*. Method *mc1* is also inherited to class *c2*. Should this contribute to coupling between classes *c2* and *d1*? And what if an inherited method is overridden, as in the case of *mc2* of *c2*. Eder *et al.* state that coupling between classes requires the invoking method to be implemented in the client class. Applied to our example, it follows that *mc1* declared in *c2* does not contribute to coupling of *c2*, whereas *mc2*, which is implemented in *c2*, does.

b) If we agree that an inherited method does not contribute to import coupling of the inheriting class, then what about an invocation *of* an inherited method? For instance, *mc3* of *c2* invokes *mc1* implemented in *c1*. The authors of all frameworks discussed here find that invoking an inherited method is special. It contributes to coupling, and has to be distinguished from invoking a method of an unrelated class. This type of coupling is commonly referred to as “inheritance-based” or “inheritance-related” coupling.

c) Due to polymorphism, one method invocation can actually access a variety of methods implemented in different classes. Consider method *mc2* of *c1* invoking *md2* of *d1*. This clearly contributes to coupling between *c1* and *d1*. However, the dynamic type of the object pointed to by *o1* may be any descendent class of *d1*. Does *mc2* therefore also contribute to coupling between *c1* and *d*2 or *d3*? Eder *et al.* state that interaction coupling requires the invoked method to be implemented in the server class. Applied to our example, method *mc2* of class *c1* contributes to coupling between *c1* and *d1*, and between *c1* and *d2*. It does not couple *c1* to class *d3*, because method *md2* is only declared in class *d3*. The rationale behind this decision is that within the framework of Eder *et al.*, all possible relationships, and thus all possible dependencies between methods should be accounted for.

d) In b) we discussed the case that a class invokes a method it inherited from its parent class. Let us now consider the case that the client and server classes are not related through inheritance. The client class can invoke a method which the server class has inherited. For instance, method *mc3* of class *c2* invokes method *md1* of *d2*. Class *d2* has inherited *md1* from *d1*. Should this therefore contribute to coupling between *c2* and *d1*? And what about the same method *mc3* invoking method *md2* of *d2*? Class *d2* inherits but overrides method *md2* of *d1*. Again, we can apply the principle of Eder *et al.* that the invoked method has to be implemented in the server class. It follows, that the invocation of *md2* contributes to coupling between *c2* and *d2* (because *md2* is implemented in *d2*). The invocation of *md1* contributes to coupling between *c2* and *d1* (because *md1* is only declared in *d2,* but implemented in *d1*).

4.2.7 Summary and conclusions

To summarize, the following about coupling in object-oriented systems is noted:

- there are different types of coupling among classes, methods, attributes
- classes and methods can be coupled more or less strongly, depending on
    1. the type of connections between them
    2. the frequency of connections between them
- a distinction can be made between import and export coupling (client-server relationships)
- both direct and indirect coupling may be relevant
- the server class can be stable or unstable
- the effect of inheritance on coupling has to be considered.

From this list we can see that there exists a variety of decisions to be made during the definition of a coupling measure. It is important that decisions are based on the intended application of the measure if the measure is to be useful. When no decision for a particular aspect can be made, all alternatives should be investigated. A second observation is that because the different aspects of coupling are independent of each other, a large number of coupling measures could be defined - this defines the problem space for coupling in object-oriented systems. In the following section, a review of object-oriented coupling measures in the software engineering literature is presented. Discussion of how existing measures address the different aspects of coupling takes place and insight is provided into how complete the overall problem space for coupling is covered by these measures.

4.3 Survey of coupling measures

The survey of coupling measures is organised as follows. Section 4.3.1 introduces the criteria for comparing the measures. A summary of existing measures according to these criteria is provided in Section 4.3.2 and conclusions are then drawn about the state-of-the-art. Sections 4.3.3 to 4.3.8 then discuss how existing coupling measures address the issues highlighted in Section 4.2. Finally, Section 4.3.9 presents a theoretical validation of the coupling measures and summarises the results of this section.

4.3.1 Criteria of comparison

In this section, we provide a list of criteria required to allow an initial comparison of measures to be performed and define the different levels of each criterion. These criteria then form the basic structure for the summary presented in Table 2.

- **Name**: The name of the measure.
- **Definition**: The definition of the measure using the defined formalism. The original definition of the measures are often ambiguous; hence, additional interpretation is required to define them using the formalism. We provide where necessary the most likely unambiguous alternative (in some cases several plausible versions of a measure are proposed).
- **Operationally defined (yes or no):** Indicates if the original definition of the measure is operational or not, i.e., was additional interpretation of the measure's original definition necessary to come up with the definition of the measure given in column "Definition".
- **Objectivity (subjective or objective):** For an objective measure, the collected measurement data does not depend on the person collecting the data, i.e., the measure is automatable. For a subjective measure, the measurement data depends on the person collecting it and hence the measure is not automatable.
- **Level of measurement** (nominal, ordinal, interval or ratio): The type of scale the measure is defined on. The type of scale is determined by the admissible transformations for the used empirical relation system [Fen91]. However, the empirical relation system used for the attribute is rarely provided. If it is not provided, the indicated scale type reflects our intuitive judgment.
- **Partially usable** (An/HLD/LLD/Imp): This column and column "Usable" address the question when, in the development process, the measures become applicable. For this purpose, a generic object-oriented development process consisting of four development phases is used: analysis, high-level design, low-level design, and implementation. Details about these development phases can be found in Appendix C.
  A measure is classed as *partially usable* at the end of a development phase if the information required for the

data collection is available at that phase, but is subject to refinement in later development phases. The column states the earliest development phase at which the measure is *partially usable*. Measurement values obtained at a development phase where the measure is *partially usable* are only approximations; their values are likely to change in subsequent development phases.

- **Usable** (An/HLD/LLD/Imp): A measure is *usable* at a given development phase if all information required for data collection is available and stable, i.e., the information is refined only to a limited extent in subsequent development phases. We state the earliest development phase at which the measure is *usable*.
- **Language specific**: If the measure is specific to a particular programming language, the language is provided. If a measure is language-specific, this does not imply that the measure is not applicable to other languages., but adapting the measure will be necessary before it can be applied to other languages.
- **Validation** (th, emp, no): Indicates if and how the measure has been validated. There is a distinction between:
  - Theoretical validation (th): The authors have validated their measure theoretically, usually by analyzing their mathematical properties. The analysis and results can be found in the first publication referenced in the "source" column (see next item on this list).
  - Empirical validation (emp): The measure has been used in an empirical validation investigating its causal relationship on an external attribute of a class, subsystem, or system. For these measures, the validation results are discussed in Section 6.0.
- **Source**: Literature references where the measure has been proposed.

4.3.2 Overview of measures

An overview of the coupling measures proposed in the literature is presented in Table 2. Before discussing the individual measures in detail a number of simple, but important observations can be made. First, there is no measure in Table 2 which is classified "usable" at the analysis or HLD phases. For measures classified "partially usable" at analysis or HLD only approximations of the values that are obtained at LLD or implementation can be computed. Empirical studies are required to analyse how accurate such approximations are and whether they are useful predictors of external attributes.

Second, the original definitions of many of the measures have been found not to be fully operational, i.e., additional interpretation has been required to formalise these measures. This is best illustrated by an example from Martin who proposes two coupling measures, Ca (afferent coupling) and Ce (efferent coupling) which use the term *category* (a set of classes that achieve some common goal) [Mar94]. Measures Ce and Ca are defined as:

- Ca for a category is the number of classes outside the category that depend upon classes within the category
- Ce for a category is the number of classes inside the category that depend upon classes outside the category.

But because Martin does not specify what causes dependencies between classes, formal definitions for Ce and Ca are impossible to provide. This is an extreme case where, although the underlying concepts are interesting, the measure is too abstract for a consistent use in practice (all other measures are at least usable with additional interpretation). It highlights the importance of a precise framework for the definition of new measures.

Third, many measures are neither validated theoretically or, more importantly, empirically. Consequently, little evidence exists to support the notion that the coupling measures are actually useful in terms of predicting relevant qualities. To strengthen software measurement research and to convince practitioners of the usefulness of software measures, a larger number of such validation studies must be performed.

Name	Definition	Operational definition	Objectivity	Scale	partly usable	usable	Language-specific	Validation	Source
CBO (coupling between object classes)	$CBO(c) = \lvert\{d \in C - \{c\} \mid uses(c,d) \vee uses(d,c)\}\rvert$	no	obj	ratio	An	Imp	no	th & emp	[CK94]
CBO´	$CBO'(c) = \lvert\{d \in C - (\{c\} \cup Ancestors(C)) \mid uses(c,d) \vee uses(d,c)\}\rvert$	no	obj	ratio	An	Imp	no	th & emp	[CK91]
RFC_α (response for class)	$RFC_\alpha(c) = \left\lvert \bigcup_{i=0}^{\alpha} R_i(c) \right\rvert$, for $\alpha = 1, 2, 3, \ldots$, where $R_0(c) = M(c)$ and $R_{i+1}(c) = \bigcup_{m \in R_i(c)} PIM(m)$	no	obj	ord	HLD	Imp	no	no	[CS95b]
RFC	$RFC(c) = RFC_1(c)$	no	obj	ord	HLD	Imp	no	th & emp	[CK94]
RFC´	$RFC'(c) = RFC_\infty(c)$	no	obj	ord	HLD	Imp	no	th	[CK91]
MPC (message passing coupling)	$MPC(c) = \sum_{m \in M_I(c)} \sum_{m' \in SIM(m) \div M_I(c)} NSI(m,m')$	no	obj	ratio	LLD	Imp	no	emp	[LH93]
DAC (data abstraction coupling)	$DAC(c) = \lvert\{a \mid a \in A_I(c) \wedge T(a) \in C\}\rvert$	no	obj	ratio	An	LLD	no	emp	[LH93]
DAC´	$DAC'(c) = \lvert\{T(a) \mid a \in A_I(c) \wedge T(a) \in C\}\rvert$	no	obj	ratio	An	LLD	no	emp	[LH93]
COF (coupling factor)	$COF(C) = \frac{\sum_{c \in C} \lvert\{d \mid d \in C \setminus (c \cup Ancestors(c)) \wedge uses(c,d)\}\rvert}{\lvert C\rvert^2 - \lvert C\rvert - \left(2 \sum_{c \in C} \lvert Desendents(c)\rvert\right)}$	no	obj	ord	An	Imp	no	no	[AGE95]
ICP (information-flow-based coupling)	$ICP^c(m) = \sum_{m' \in PIM(m) - (M_{NEW}(c) \cup M_{OVR}(c))} (1 + \lvert Par(m')\rvert) \cdot NPI(m,m')$ $ICP(c) = \sum_{m \in M_I(c)} ICP^c(m)$ $ICP(SS) = \sum_{c \in SS} ICP(c)$	yes	obj	ratio	LLD	Imp	no	th	[LLWW95]

Table 2. Coupling measures

Name	Definition	Operational definition	Objectivity	Scale	partly usable	usable	Language-specific	Validation	Source
NIH-ICP (information-flow-based non-inheritance coupling)	$NIH\text{–}ICP^c(m) = \sum_{m' \in R} (1 + \lvert Par(m') \rvert) \cdot NPI(m, m')$, where $R = PIM(m) \cap (\bigcup_{c' \in Anc(c)} M(c'))$ $NIH\text{–}ICP(c) = \sum_{m \in M_I(c)} NIH\text{–}ICP^c(m)$ $NIH\text{–}ICP(SS) = \sum_{c \in SS} NIH\text{–}ICP(c)$	yes	obj	ratio	LLD	Imp	no	th	[LLWW95]
IH-ICP (information-flow-based inheritance coupling)	$NIH\text{–}ICP^c(m) = \sum_{m' \in R} (1 + \lvert Par(m') \rvert) \cdot NPI(m, m')$, where $R = PIM(m) \cap (\bigcup_{c' \in C - (\{c\} \cup Ancestors(c))} M(c'))$ $IH\text{–}ICP(c) = \sum_{m \in M_I(c)} IH\text{–}ICP^c(m)$ $IH\text{–}ICP(SS) = \sum_{c \in SS} IH\text{–}ICP(c)$	yes	obj	ratio	LLD	Imp	no	th	[LLWW95]
IFCAIC	$IFCAIC(c) = \lvert \{a \mid a \in A_I(c) \wedge T(a) \in Friends^{-1}(c)\} \rvert$	yes	obj	ratio	An	LLD	C++	th & emp	[BDM96]
ACAIC	$ACAIC(c) = \lvert \{a \mid a \in A_I(c) \wedge T(a) \in Ancestors(c)\} \rvert$	yes	obj	ratio	An	LLD	no	th & emp	[BDM96]
OCAIC	$OCAIC(c) = \lvert \{a \mid a \in A_I(c) \wedge T(a) \in (Others(c) \cup Friends(c))\} \rvert$, where $Others(c) = C - (Ancestors(c) \cup Desendents(c) \cup Friends(c) \cup Friends^{-1}(c) \cup \{c\})$	yes	obj	ratio	An	LLD	C++	th & emp	[BDM96]
FCAEC	$FCAEC(c) = \sum_{c' \in Friends(c)} \lvert \{a \mid a \in A_I(c') \wedge T(a) = c\} \rvert$	yes	obj	ratio	An	LLD	C++	th & emp	[BDM96]
DCAEC	$DCAEC(c) = \sum_{c' \in Descendents(c)} \lvert \{a \mid a \in A_I(c') \wedge T(a) = c\} \rvert$	yes	obj	ratio	An	LLD	no	th & emp	[BDM96]

Table 2. Coupling measures

Name	Definition	Operational definition	Objectivity	Scale	partly usable	usable	Language-specific	Validation	Source
OCAEC	$OCAEC(c) = \sum_{c' \in Others(c) \cup Friends^{-1}(c)} \lvert\{a \mid a \in A_I(c') \wedge T(a) = c\}\rvert$	yes	obj	ratio	An	LLD	C++	th & emp	[BDM96]
IFCMIC	$IFCMIC(c) = \sum_{m \in M_{NEW}(c)} \lvert\{a \mid a \in Par(m) \wedge T(a) \in Friends^{-1}(c)\}\rvert$	yes	obj	ratio	HLD	LLD	C++	th & emp	[BDM96]
ACMIC	$ACMIC(c) = \sum_{m \in M_{NEW}(c)} \lvert\{a \mid a \in Par(m) \wedge T(a) \in Ancestors(c)\}\rvert$	yes	obj	ratio	HLD	LLD	no	th & emp	[BDM96]
OCMIC	$OCMIC(c) = \sum_{m \in M_{NEW}(c)} \lvert\{a \mid a \in Par(m) \wedge T(a) \in Others(c) \cup Friends(c)\}\rvert$	yes	obj	ratio	HLD	LLD	C++	th & emp	[BDM96]
FCMEC	$FCMEC(c) = \sum_{c' \in Friends(C)} \sum_{m \in M_{NEW}(c')} \lvert\{a \mid a \in Par(m) \wedge T(a) = c\}\rvert$	yes	obj	ratio	HLD	LLD	C++	th & emp	[BDM96]
DCMEC	$DCMEC(c) = \sum_{c' \in Descendents(c)} \sum_{m \in M_{NEW}(c')} \lvert\{a \mid a \in Par(m) \wedge T(a) = c\}\rvert$	yes	obj	ratio	HLD	LLD	no	th & emp	[BDM96]
OCMEC	$OCMEC(c) = \sum_{c' \in Others(c) \cup Friends^{-1}(c)} \sum_{m \in M_{NEW}(c')} \lvert\{a \mid a \in Par(m) \wedge T(a) = c\}\rvert$	yes	obj	ratio	HLD	LLD	C++	th & emp	[BDM96]
OMMIC	$OMMIC(c) = \sum_{m \in M_I(c)} \sum_{m' \in R} (NSI(m,m') + PP(m,m'))$, where $R = SIM(m) \cap (\bigcup_{c' \in Others(c)} M(c') \cup \bigcup_{c' \in Friends(c)} M(c'))$	yes	obj	ratio	LLD	Imp	C++	th & emp	[BDM96]
IFMMIC	$IFMMIC(c) = \sum_{m \in M_I(c)} \sum_{m' \in R} (NSI(m,m') + PP(m,m'))$, where $R = SIM(m) \cap \left(\bigcup_{c' \in Friends^{-1}(c)} M(c')\right)$	yes	obj	ratio	LLD	Imp	C++	th & emp	[BDM96]
AMMIC	$AMMIC(c) = \sum_{m \in M_I(c)} \sum_{m' \in R} (NSI(m,m') + PP(m,m'))$, where $R = SIM(m) \cap (\bigcup_{c' \in Ancestors(c)} M(c'))$	yes	obj	ratio	LLD	Imp	no	th & emp	[BDM96]

Table 2. Coupling measures

Name	Definition	Operational definition	Objectivity	Scale	partly usable	usable	Language-specific	Validation	Source
OMMEC	$OMMEC(c) = \sum_{c' \in Others(c) \cup Friends^{-1}(c)} \sum_{m' \in M_I(c)} \sum_{m \in R} (NSI(m', m) + PP(m', m))$ where $R = SIM(m') \cap (M_{NEW}(c) \cup M_{OVR}(c))$	yes	obj	ratio	LLD	Imp	C++	th & emp	[BDM96]
FMMEC	$FMMEC(c) = \sum_{c' \in Friends(c)} \sum_{m' \in M_I(c)} \sum_{m \in R} (NSI(m', m) + PP(m', m))$, where $R = SIM(m') \cap (M_{NEW}(c) \cup M_{OVR}(c))$	yes	obj	ratio	LLD	Imp	C++	th & emp	[BDM96]
DMMEC	$DMMEC(c) = \sum_{c' \in Ancestors(c)} \sum_{m' \in M_I(c)} \sum_{m \in R} (NSI(m', m) + PP(m', m))$, where $R = SIM(m') \cap (M_{NEW}(c) \cup M_{OVR}(c))$	yes	obj	ratio	LLD	Imp	no	th & emp	[BDM96]

Table 2. Coupling measures

4.3.3 Types of coupling

Note that many measures in Table 2 are based on method invocations and attributes references. All versions of RFC, MPC, and the ICP family by Lee *et al.* are based solely on method invocations. The "method-method interaction" measures OMMIC, IFMMIC, AMMIC and OMMEC, FMMEC and DMMEC count method invocations plus occurrences where a method is passed a pointer to another method. And measures CBO and COF include references to both methods and attributes.

DAC, DAC´ and the "class-attribute interaction" measures IFCAIC, ACAIC, OCAIC and FCAEC, DCAEC and OCAEC are measures which take into account aggregation and could be classified as "component coupling" according to the framework by Eder *et al.* The "class-method interaction" measures IFCMIC, ACMIC, OCMIC and FCMEC, DCMEC and OCMEC in the metrics suite by Briand *et al.* also measure "component coupling". These measures count occurrences where a parameter of a method has another class as its type.

Comparison of the types of coupling used by the measures in Table 2 to those introduced in the coupling frameworks in Table 1 shows that all types of coupling used by the measures are present in at least one framework. In contrast, however, there are types of coupling in Table 1 for which there are no measures defined, namely, type #7 (a method of a class *c* has a local variable of class *d),* and type #8 (a method of a class *c* invokes a method while passing an object of type class *d*). These types of coupling may be less interesting because they typically are available only after implementation. In conclusion, there are types of coupling in object-oriented systems which have not yet been considered - empirical studies are required to investigate the usefulness of these types.

4.3.4 Strength of coupling

In this section, we first examine how different measures account for different strengths of coupling. We then examine how the measures deal with the frequency of coupling connections.

Measures CBO and COF do not distinguish between method invocations and attribute references; both types of coupling are treated as one and the same. The "method-method interaction" measures by Briand *et al.* do not distinguish between method invocation and passing a pointer to a method *m'* as a parameter to some other method *m* (because *m* can then invoke *m'*). All other measures focus only on one type of coupling. Incorporating more than one type of coupling into a single measure may be questionable and needs to be clearly justified: it requires relative strengths to be assigned to the different types of coupling under consideration (e.g., the measures just mentioned treat different types of connections as one and the same). Such an assignment is subjective and makes empirical validation difficult. Questions will arise such as "What type of coupling contributed how much to the measurement values?" To answer these questions, the types of coupling have to be measured separately. So, unless there is a clear justification, it is strongly recommended not to combine different types of coupling. In addition, the assignment of relative strengths of coupling will depend on the measurement goal and other environmental factors (e.g., design methodology, programming language). So assigning strengths to the various types of coupling should be part of a prediction model for some external attribute (e.g., a model that predicts fault-proneness of a class from different types of coupling), but not part of the definition of a measure.

"Connection" is a generic term defined as an occurrence of a given type of coupling (e.g., a method invocation or an attribute having a class as its type). The measures CBO and COF take a binary approach to coupling between classes: two classes are either coupled or not and the number of such "class couples" is counted. However, this approach does not make use of all the information available. For instance, measure CBO has been proposed by Chidamber and Kemerer as an indicator for maintainability, testability and reusability of a class [CK94]. However, the maintainability and testability of the class is also likely to be influenced by the frequency of connections between coupled classes and not only by the number of classes to which it is coupled. For example, a class *c* which is loosely coupled to (i.e., has few connections to) five other classes may be easier to maintain or test than a class *d* which is strongly coupled (i.e., has many connections) to only two other classes. All other measures include individual connections between classes. RFC counts the number of methods invoked by a class. MPC and the "method-method interaction" measures by Briand *et al.* count the number of method invocations. The ICP measures also take the number of parameters passed to each method into account (within the framework by Hitz and Montazeri this is the factor "Complexity of interface" for object level coupling). DAC and the measures for class-attribute and class-method interaction by Briand *et al.* count the number of attributes and parameters having a class type. DAC´ counts the number of classes used as a type for an attribute.

An important difference is between the “number of methods invoked” and the “number of method invocations”. If the same method is invoked more than once and each invocation is counted separately, a distorted value for the measure can arise. Consider two extreme cases:

- Class c_1 invokes a method *m* ten times.
- Class c_2 invokes ten different methods of ten different classes, once each.

For the MPC measure (and other measures counting method invocations), MPC(c_1)=MPC(c_2)=10. In reality however, the coupling of class c_2 could be considered worse than that of c_1 because c_2 is coupled to 10 different classes whereas c_1 is coupled to only one.

Similar distorted values arise for attributes. Contrast measure DAC which counts the number of attributes having a class as its type with DAC´ which counts the number of classes used as types of attributes. If a class c_1 has ten attributes of type class c_2, DAC(c_1)=10 whereas DAC´(c_1)=1.

At this point, it can be concluded that there are several ways to take into account the frequencies of connections between classes. They all have their strengths and weaknesses, and they all can be justified by an empirical relational system. That is, there is not “one right way” to count frequencies of connections. How to count them must be decided with respect to a given measurement goal.

4.3.5 Import and export coupling

CBO makes no distinction between import and export coupling: two classes are coupled if one uses the other or vice versa. COF distinguishes the cases where a class c_1 uses a class c_2, and class c_2 uses class c_1. In the case where both classes use each other both relationships will be counted separately. The suite by Briand *et al.* provides separate measures for import and export coupling (e.g., OMMEC & OMMIC). All other measures only consider import coupling (i.e., the role of the class as client).

4.3.6 Direct and indirect coupling

Most of the coupling measures consider direct coupling only. RFC´ is the number of methods that can possibly be invoked by sending a message to a class *c*. This includes methods of *c*, methods invoked by the methods of *c*, the methods these in turn invoke, and so on. In that sense, indirect coupling is accounted for. RFC_α counts such nested method invocations up to a specified level α.

We can easily derive new measures that account for indirect coupling from measures that do not (if it appears sensible to do so). Direct coupling describes a relation on a set of elements (e.g., a relation “invokes” on the set of all methods of the system, or a relation “uses” on the set of all classes of the system). To account for indirect coupling, we need only use the transitive closure of that relation.

4.3.7 Stability of server class

This aspect is not addressed by any of the proposed coupling measures. A possible application would be to distinguish between coupling among problem domain classes and import coupling from classes taken from standard libraries (or any other classes that are not subject to development or change in the ongoing project).

A pragmatic method of identifying the stability of server classes without defining new measures is to use an existing import coupling measure and measure, for a given class, its import coupling from problem domain classes and library classes separately. Thus it can be determined to what extent the class relies on problem domain classes and library classes.

4.3.8 Inheritance

The distinction between inheritance-based and non-inheritance based coupling can be found frequently in the literature. Inheritance-based coupling refers to coupling between a class and its ancestors. Non-inheritance based coupling is coupling between two classes with no inheritance relationship between them. For instance, the design principle by Coad and Yourdon suggests to maximise inheritance-based coupling and minimise non-inheritance-based coupling [CY91a], [CY91b]). Excluding inheritance-based coupling corresponds with the point of view that

the inheriting class “has” the attributes and methods it inherited and using an inherited method or attribute is not equivalent to coupling with another class. Including inheritance-based coupling corresponds with the point of view that the inheriting class does not “have” the inherited attributes and methods. This view is supported by the fact that some programming languages restrict access to inherited methods and attributes (e.g., private methods and attributes in a C++ class cannot be used by its children classes). Empirical studies are required to realise if and how inheritance-based and non-inheritance-based coupling should be distinguished. It is also conceivable that the relative importance of both types of coupling depends on the respective measurement goal: what is empirically justified in one situation may not apply in other situations.

CBO' measures “non-inheritance based coupling” [CK91] whereas CBO explicitly includes “coupling due to inheritance” [CK94]. Chidamber and Kemerer do not explain why they first excluded inheritance-based coupling and why they introduced it later. Similarly, the definition of COF excludes inheritance-based coupling without any explanation [AGE95].

Lee *et al.* acknowledge the need to differentiate between inheritance-based and non-inheritance-based coupling by proposing corresponding measures: NIH-ICP counts non-inheritance-based coupling only, IH-ICP counts inheritance-based coupling only. ICP is the sum of IH-ICP and NIH-ICP, thus treats both types of coupling equal. The suite of measures by Briand *et al.* also provides measures which count inheritance-based and non-inheritance based coupling separately. Unlike Lee *et al.*, they do not define a single measure which counts both types of coupling (cf. the above discussion on types of coupling). All other measures do not address inheritance. The definitions of these measures are therefore ambiguous and have been marked “not operationally defined” in Table 2.

For most measures, the other inheritance issues introduced in Section 4.2.6 (how to account for polymorphism, overriding of methods, virtual methods etc.) are not considered in the original definitions. [EKS94] is the only publication we are aware of that describes these issues in detail.

Lee *et al.* emphasize that their ICP measures, which are based on method invocations, take polymorphism into account [LLWW95]. Their measures aim at measuring the amount of information flow between methods. Information flow occurs between a method *m* and any method m' that is possibly invoked by *m* which includes methods $m' \in PIM(m)$. The measures for method-method-interaction by Briand *et al.* are counts of static method invocations only [BDM96].

4.3.9 Theoretical validation

In this section, we theoretically validate the coupling measures in Table 2 with respect to five coupling properties defined by Briand *et al.* [BMB96]. The motivation behind defining such properties is that a measure must be supported by some underlying theory - if it is not, then the usefulness of that measure is questionable. The five coupling properties defined by Briand *et al.* are one of the more recent proposals to characterize coupling in a reasonably intuitive manner. While these properties are not sufficient to say that a measure which fulfils them all will be useful, it is likely that the reverse statement will be true.

The coupling properties were originally defined using a generic, mathematical framework to model a software system. For the purpose of validating coupling measures for object-oriented systems, we adapt this framework to model an object-oriented system as defined by the formalism in Section 3.0.

In the following discussion let *Coupling* be a candidate measure for coupling of a class or an object-oriented system. Relationships capture the connections between classes the respective coupling measure is focused on. As the coupling measure can measure import or export coupling (or both), *OuterR(c)* will denote the relevant set of relationships from or to class *c* (or both). We define $InterR(C) = \bigcup_{c \in C} OuterR(c)$ to be the set of inter-class relationships in system *C*.

The five proposed coupling properties are:

Coupling.1: Nonnegativity. The coupling [of a class *c* | of an object-oriented system *C*] is nonnegative:

$$[\ Coupling(c) \geq 0 \quad | \quad Coupling(C) \geq 0\ ]$$

Coupling.2: Null value. The coupling [of a class *c* | of an object-oriented system *C*] is null if [*OuterR(c)* | *InterR(C)*] is empty:

$$[\ OuterR(c) = \varnothing \Rightarrow Coupling(c) = 0 \quad | \ InterR(C) = \varnothing \Rightarrow Coupling(C) = 0\ ]$$

Coupling.3: Monotonicity. Let *C* be an object-oriented system and $c \in C$ be a class in a *C*. We modify class *c* to form a new class c' which is identical to *c* except that $OuterR(c) \subseteq OuterR(c')$, i.e., we added some relationships to *c*. Let C' be the object-oriented system which is identical to *C* except that class *c* is replaced by class c'. Then

$$[\ Coupling(c) \leq Coupling(c') \quad | \quad Coupling(C) \leq Coupling(C')\ ]$$

Coupling.4: Merging of classes. Let *C* be an object-oriented system, and $c_1, c_2 \in C$ two classes in *c*. Let c' be the class which is the union of c_1 and c_2. Let C' be the object-oriented system which is identical to *C* except that classes c_1 and c_2 are replaced by c'. Then

$$[\ Coupling(c_1) + Coupling(c_2) \geq Coupling(c') \quad | \quad Coupling(C) \geq Coupling(C')\ ]$$

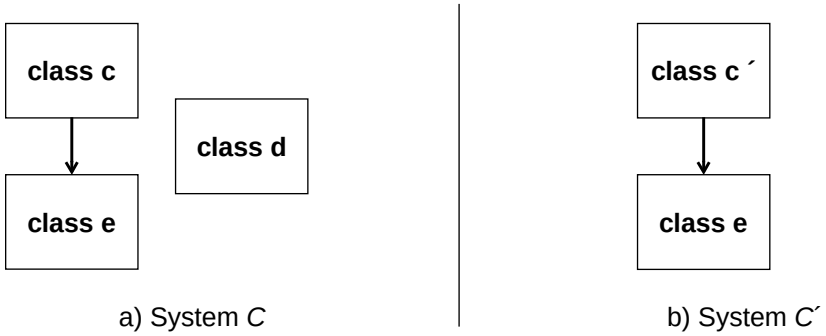
Coupling.5: Merging of unconnected classes. Let *C* be an object-oriented system, and $c_1, c_2 \in C$ two classes in *c*. Let c' be the class which is the union of c_1 and c_2. Let C' be the object-oriented system which is identical to *C* except that classes c_1 and c_2 are replaced by c'. If no relationships exist between classes c_1 and c_2 in *C*, then

$$[\ Coupling(c_1) + Coupling(c_2) = Coupling(c') \quad | \quad Coupling(C) = Coupling(C')\ ]$$

Coupling.3 specifies that if a relationship is added to the system, coupling must not decrease. Coupling.4 specifies that merging two classes must not increase coupling because relationships disappear (namely those between the classes that have been merged). Coupling.5 specifies that merging two unconnected classes must not affect coupling at all.

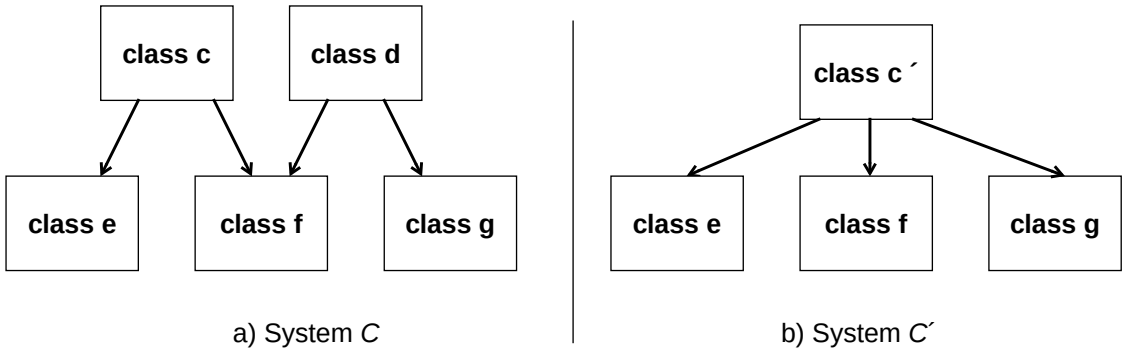
In the following we discuss which measures violate one or more of the coupling properties above.

- RFC_α and its special cases RFC and RFC´ do not have a null value (Coupling.2): If *c* is a class with five methods which do not invoke any other methods, we have $RFC_\alpha(c)=5$, even though *OuterR*(*c*)=*0*.
- measures DAC and DAC´ also do not have a null value (Couping.2). If class *c* has an attribute of type class *c* (or, more realistically, an attribute of type “pointer to class *c*”), then DAC(*c*) > 0 and DAC´(*c*) > 0, even though *OuterR*(*c*)=*0*.
- COF violates Coupling.4 and Coupling.5: Consider the example systems in Figure 2. Boxes are classes, the arrow from class *c* to class *e* indicates that class *c* uses class *e*. For system *C*, we have COF(*C*)=1/6. In system *C´*, classes *c* and *d* have been merged to form a new class *c´*. We have COF(*C´*) = 1/2, and thus COF(*C*) < COF(*C´*). Therefore, Coupling.4 and Coupling.5 are violated. This is due to the fact that COF has been normalized to range between [0,1].



a) System *C* b) System *C´*

FIGURE 2. Counterexample for COF



a) System *C* b) System *C´*

FIGURE 3. Counterexample for CBO

- CBO and CBO´ do not fulfil Coupling.5. Consider the example system in Figure 3. The semantics of the symbols is the same as in Figure 2. For system *C*, we have CBO(*c*)=CBO(*d*)=2. In system *C´*, classes *c* and *d* have been merged to form a new class *c´*. It is CBO(*c´*)=3, thus Coupling.5 is violated.
- Measure DAC´ does not fulfil Coupling.5. To see this, we reinterpret the meaning of the arrows in Figure 3: the arrow from class *c* to class *d* now indicates that class *c* has one or more attributes of type class *d*. Then, we have DAC´(*c*)=DAC´(*d*)=2 for system *C*, but DAC´(*c´*)=3 for system *C´*, and Coupling.5 is violated. Measure DAC fulfils Coupling.5. We always have DAC(*c*)+DAC(*d*)=DAC(*c´*), because this measure counts the number of attributes having a class as their type.
- Likewise, RFC, RFC´ and RFC_α violate Coupling.5. Consider the example in Figure 4. We have RFC(c)=RFC(d)=2 (each class has one method and invokes one other method, RFC and RFC_α yield the same values). If we merge classes *c* and *d* to form a new class *c´*, we get RFC(*c´*)=3 (*class c´* has two methods and calls one other method).

```
class e{
     me(int i) { /*...*/};
};
```

```
class c{
     e e_obj1;
     mc() { e_obj1.me(1) }
};
class d{
     e e_obj2;
     md() { e_obj2.me(2) }
};
```

FIGURE 4. Counterexample for RFC

There is a pattern visible concerning the violations of property Coupling.5. Measures CBO, CBO´, the RFC measures and DAC´ all have in common that multiple connections to the same method or class are counted as one. If two unconnected classes *c* and *d* use a third class *e* in common, and we merge classes *c* and *d* to form a new class *c´*, then the previously two connections to class *e* are only counted as one connection from new class *c´* to class *e*. As a result, the coupling of the new class *c´* is lower than the sum the of the coupling of classes *c* and *d*. Thus, property Coupling.5 only holds for measures which count individual connections.

The ICP “information-flow-based” measures fulfil all five coupling properties, but show a special behaviour. They measure the amount of information flowing in to and out from the class via parameters through method invocation, i.e., the measures sum the number of parameters (plus one) passed at each method invocation. Consequently, invoking, say, six methods with no parameters contributes the same to class coupling as invoking two methods each with two parameters or invoking one method with five parameters. This implies a special empirical model - weighting method invocations by the number of parameters passed should be investigated empirically.

Table 3 summarizes the results of this section. For each measure, we indicate the type of coupling it uses, what factors determine the strength of coupling, if it is an import or export coupling measure, if indirect coupling is accounted for, and how inheritance is dealt with (inheritance-based coupling, non-inheritance-based coupling, or both). The columns C2, C4 and C5 show violations of the coupling properties Coupling.2, Couping.4 and Coupling.5, where an “X” indicates a violation (properties Coupling.1 and Coupling.3 are fulfilled by all measures and therefore are not listed in the table).

Measure	Type of coupling	Strength of coupling	Import or export coupling	Indirect coupling	Inheritance	C2	C4	C5
CBO	method invocation, attribute reference	#coupled classes	both (indifferent)	no	both			x
CBO´					non-inheritance based			x
RFC_{α}	method invocation	#methods invoked	import	depends	both	x		x
RFC				no		x		x
RFC´				yes		x		x
MPC	method invocation	#method invocations	import	no	both			
DAC	type of attribute	#attributes	import	no	class type may be ancestor, inherited attributes not considered	x		
DAC´		# distinct types	import	no		x		x
COF	method invocation, attribute reference	#coupled classes	both (distinguished)	no	non-inheritance-based		x	x
ICP	method invocation	#method invocations, #parameters passed	import	no	both			
IH-ICP					inheritance-based			
NIH-ICP					non-inheritance-based			
IFCAIC	type of attribute	#attributes	import	no	non-inheritance-based			
ACAIC			import	no	inheritance-based			
OCAIC			import	no	non-inheritance-based			
FCAEC			export	no	non-inheritance-based			
DCAEC			export	no	inheritance-based			
OCAEC			export	no	non-inheritance-based			

Table 3: Properties of coupling measures

Measure	Type of coupling	Strength of coupling	Import or export coupling	Indirect coupling	Inheritance	C2	C4	C5
IFCMIC	type of parameter	#of parameters	import	no	non-inheritance-based			
ACMIC			import	no	inheritance-based			
OCMIC			import	no	non-inheritance-based			
FCMEC			export	no	non-inheritance-based			
DCMEC			export	no	inheritance-based			
OCMEC			export	no	non-inheritance-based			
OMMIC	method invocation, passing of pointer to method	#method invocations, #pointers passed to a method	import	no	non-inheritance-based			
IFMMIC			import	no	non-inheritance-based			
AMMEC			import	no	inheritance-based			
OMMEC			export	no	non-inheritance-based			
FMMEC			export	no	non-inheritance-based			
DMMEC			export	no	inheritance-based			

Table 3: Properties of coupling measures

5.0 A Unified Framework for Coupling Measurement

In this section, a new framework for coupling in object-oriented systems is proposed. The framework is defined on the basis of the issues identified by comparing the coupling frameworks (see Section 4.2) and the discussion of existing measures with respect to these issues (see Sections 4.3.3 to 4.3.8). The objective of the unified framework is to support the comparison and selection of existing coupling measures with respect to a particular measurement goal. In addition, the framework should provide guidelines to support the definition of new measures with respect to a particular measurement goal when there are no existing measures available. The framework, if used as intended, will

- ensure that measure definitions are based on explicit decisions and well understood properties,
- ensure that all relevant alternatives have been considered for each decision made,
- highlight dimensions of coupling for which there are few or no measures defined.

The framework consists of six criteria, each criterion determining one basic aspect of the resulting measure. First, we describe each criterion: what decisions have to be made, what are the available options, how is the criterion reflected by the coupling measures in Section 4.3. We then discuss how the framework can be used to derive coupling measures. For each criterion, we have to choose one or more of the available options which will be strongly influenced by the stated measurement goal. Finally, we provide a set of guidelines how a given measurement goal influences this choice.

The six criteria of the framework are:

1. The type of connection, i.e., what constitutes coupling.
2. The locus of impact, i.e., import or export coupling.
3. Granularity of the measure: the domain of the measure and how to count coupling connections.
4. Stability of server.
5. Direct or indirect coupling.
6. Inheritance: inheritance-based vs. non-inheritance-based coupling, and how to account for polymorphism, and how to assign attributes and methods to classes.

These criteria are necessary to consider when specifying a coupling measure. However, they are not sufficient, as other aspects such as properties of measures (e.g., those proposed in [BMB96]) and results from empirical validation studies have to be considered too. The influence of these aspects is not addressed here.

We now describe each of the criteria in the order given above.

5.1 Framework criteria

5.1.1 Type of connection

Choosing a type of connection implies choosing the mechanism that constitutes coupling between two classes. Table 4 summarizes the possible types of connections, i.e., links between a client and a server “item” (attribute, method, or class), and those used by the reviewed measures. The items are listed in the columns “client item” and “server item”. Column “Description” explains the type of connection. Column “Design phase” indicates from which design phase on the type of connection typically is applicable. The numbers in column “#” are used later to reference the types of connections. Column “Measures” lists for each type of connection, which coupling measures use that type of connection.

#	Client item	Server Item	Description	Design phase	Measures
1	attribute *a* of a class *c*	class *d*, $d \neq c$	class *d* is the type of *a*	HLD	DAC, DAC´, IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC
2	method *m* of a class *c*	class *d*, $d \neq c$	class *d* is the type of a parameter of *m*, or the return type of *m*	HLD	IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC
3	method *m* of a class *c*	class *d*, $d \neq c$	class *d* is the type of a local variable of *m*	LLD	-
4	method *m* of a class *c*	class *d*, $d \neq c$	class *d* is the type of a parameter of a method invoked by *m*	LLD	-
5	method *m* of a class *c*	attribute *a* of a class *d*, $d \neq c$	*m* references *a*	HLD	CBO, CBO´, COF
6	method *m* of a class *c*	method m' of a class *d*, $d \neq c$	*m* invokes m'	HLD	CBO, CBO´, RFC_{α}, RFC, RFC´, MPC, COF, ICP, NIH-ICP, IH-ICP, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC
7	class *c*	class *d*, $d \neq c$	high-level relationships between classes, such as “uses”, “consists-of”	An	-

Table 4: Types of connection

5.1.2 Locus of impact

It has to be decided whether to count import or export coupling:

- Import coupling analyses attributes, methods, or classes in their role as clients (users) of other attributes, methods, or classes.
- Export coupling analyses the attributes, methods, and classes in their role as servers to other attributes, methods, or classes.

Table 5 shows which coupling measures are import coupling measures and which are export coupling measures.

Direction	Measures
Import	CBO, CBO´, RFC_{α}, RFC, RFC´, MPC, DAC, DAC´, COF, ICP, IH-ICP, NIH-ICP, IFCAIC, ACAIC, OCAIC, IFCMIC, ACMIC, OCMIC, IFMMIC, AMMIC, OMMIC
Export	CBO, CBO´, COF, FCAEC, DCAEC, OCAEC, FCMEC, DCMEC, OCMEC, OMMEC, FMMEC, DMMEC

Table 5: Import and export coupling measures

5.1.3 Granularity

The granularity of the measure is the level of detail at which information is gathered. The granularity of the measure is determined by two factors:

- the domain of the measure, i.e., what components are to be measured
- how exactly the connections are counted.

Domain of measure

Table 6 shows possible domains for the coupling measures and which measures from Section 4.3 have that domain. Of course, measures for smaller domains such as the class level can easily be extended to larger domains like the sets of classes and the system level.

Domain	Measures
attribute	-
method	ICP, IH-ICP, NIH-ICP
class	CBO, CBO´, RFC_{α}, RFC, RFC´, MPC, DAC, DAC´, ICP, IH-ICP, NIH-ICP, IFCAIC, ACAIC, OCAIC, FCAEC, DCAEC, OCAEC, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC
set of classes	ICP, IH-ICP, NIH-ICP
system	COF

Table 6: Mapping of measures to domains

How to count connections

The next decision is how to count connections. Available options for this decision can be restricted by the domain of the measure. For measures defined at the method or attribute level, two options are listed in Table 7. Column "#" provides a number used for reference purposes, and "Description" explains the option. Columns "Import coupling example" and "Export coupling example" illustrate the options using the example of references to attributes by methods (connection type 5).

#	Description	Import coupling example	Export coupling example
A)	count individual connections	for each method, the number of references to attributes	for each attribute the number of references to the attribute
B)	count the number of distinct items at the other end of the connections	for each method, the number of attributes referenced	for each attribute the number of methods that reference the attribute

Table 7: Options for counting connections at the attribute and method level

The difference between options A) and B) is that multiple connections between two items are counted separately in option A), and counted as one in option B).

At the class level, there are four options to count connections:

#	Description	Import coupling example	Export coupling example
C)	add up the number of connections counted as in A) for each method or attribute of the class	the total number of attribute references by methods in the class	the total number of references to attributes of the class
D)	add up the numbers of connections counted as in B) for each method or attribute of the class	add up the number of attributes referenced by each method of the class	add up or for each attribute of the class: the number of methods that reference the attribute
E)	count the number of distinct items at the end of connections starting from or ending in methods or attributes of the class	the number of attributes referenced by the methods of the class	the number of methods referencing attributes of the class
F)	for a class *c*, count the number of other classes to which there is at least one connection	the number of classes which have an attribute that is referenced by a method of class *c*	the number of classes which have a method that reference an attribute of class *c*

Table 8: Options for counting connections at the class level

The difference between options D) and E) is that if, for instance, two methods of a class reference the same attribute, the references are counted separately (once for each method) according to D), and counted as one for the class according to E).

In Table 8, we use phrases such as “the methods of the class” or “the attributes of the class”. This is imprecise, because it is not yet specified if a method or attribute has to be declared or implemented in the class in order to belong to it. This issue will be discussed in criterion 6 (inheritance).

Measures defined for sets of classes or the system can be constructed by adding up the number of connections of the relevant classes, counted according to one of the options C) to F).

Option A) produces measures at the finest grain whereas option F) produces measures at the coarsest grain.

Table 9 shows how coupling measures count frequencies of connections.

Option #	Measures
A)	ICP, IH-ICP, NIH-ICP
B)	-
C)	MPC, DAC, ICP, IH-ICP, NIH-ICP, IFCMIC, ACMIC, OCMIC, FCMEC, DCMEC, OCMEC, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC
D)	DAC´
E)	RFC_{α}, RFC, RFC´
F)	CBO, CBO´, COF

Table 9: Mapping of measures to options for counting the frequency of connections

5.1.4 Stability of server

For the discussion of this criterion, two different categories of class stability are defined:

- *unstable classes*: these are classes which are subject to development or modification in the project at hand. Unstable classes are problem domain classes which are being developed exclusively for the system, or are being adapted from other systems (reuse with modification)
- *stable classes*: classes that are not subject to change in the project at hand. Stable classes are classes imported from libraries, or classes reused verbatim from other systems.

Using a class (the server class) which is unstable is different from using a stable server class. If an unstable server class is modified, this may require the using class to be modified as well. This modification in turn may trigger other modifications, and so on. Since a stable server class is not subject to modification, it can not trigger an avalanche of changes that cascade through the system.

Other categorizations than the above are conceivable (e.g., verbatim reused classes, classes were few changes are expected, classes were many changes are expected, etc.). However, we suggest to use a categorization scheme where the decision, into which category a given class belongs, can be made automatically. Otherwise, the resulting coupling measures may no longer be automatically collectable.

Using the above categorization, we have basically four options to distinguish stability of the server class. These are summarized in Table 10.

Class stability has not been addressed in the definition of any of the reviewed measures. In other words, the measures make no distinction of the stability of the server class, i.e., option III.

5.1.5 Direct or indirect connections

We have to decide whether to count direct connections only or also indirect connections. For example, if a method m_1 invokes a method m_2, which in turn invokes a method m_3, we can say that m_1 indirectly invokes m_3. Methods m_1 and m_3 are indirectly connected.

RFC_{α} and RFC´ are the only measures to take indirect connections into account. All other measures count direct connections only.

Option #	Description
I	Take only connections from or to unstable classes into account, do not count connections from or to stable classes.
II	Take only connections from or to stable classes into account.
III	Take all connections into account, regardless of the stability of the class at the other end, i.e., do not distinguish stability at all.
IV	Count separately connections to stable and unstable classes. This results in two sets of measures. For example, on the class level, we would have two coupling values for each class.

Table 10: Options to account for stability of server class

5.1.6 Inheritance

Three aspects need to be considered with respect to inheritance:

- Is there a need to distinguish between inheritance-based coupling and non-inheritance based coupling?
- How do we assign methods and attributes to classes?
- For method invocations: shall we consider static or polymorphic invocations?

The aspects should be dealt with in the order they are listed.

Inheritance-based vs. non-inheritance-based coupling

First, we have to decide whether to count inheritance-based coupling and/or non-inheritance-based coupling. Inheritance-based coupling analyses connections between classes that are related via inheritance. Likewise, non-inheritance-based coupling refers to connections between classes that are not related via inheritance. We have four options for dealing with inheritance, which are described in the following table. In column “measures”, we also list which measures in Section 4.3 conforms to the respective option.

Option #	Description	Measures
I	count inheritance-based coupling only	IH-ICP, ACAIC, DCAEC, ACMIC, DCMEC, AMMEC, DMMEC
II	count non-inheritance-based coupling only	CBO´, COF, NIH-ICP, IFCAIC, OCAIC, FCAEC, OCAEC, IFCMIC, OCMIC, FCMEC, OCMEC, IFMMIC, OMMEC, FMMEC
III	count inheritance-based and non-inheritance-based coupling separately	-
IV	count inheritance-based and non-inheritance-based coupling, making no distinction	CBO, RFC_{α}, RFC, RFC´, MPC, DAC, DAC´, ICP

Table 11: Options for inheritance-based coupling

There are no measures for option III, because a single measure cannot count both inheritance-based and non-inheritance-based coupling separately (Option III). To implement this option, pairs of measures are needed, where one measure of each pair conforms to option I, the other to option II (for instance, IH-ICP and NIH-ICP).

Polymorphism

The next question is how to deal with polymorphism. This is relevant for method invocations only (connections of type 6). There are two options:

- Account for polymorphism, i.e., for a method *m*, we count connections between *m* and all methods $m' \in PIM(m)$.
- Do not account for polymorphism, i.e., for a method *m*, we count connections between *m* and methods $m' \in SIM(m)$ only.

Table 12 shows which measures in Section 4.3 account for polymorphism and which do not. Only measures that are concerned with method invocations are considered.

Type	Measure
account for polymorphism	CBO, CBO´, RFC_{α}, RFC, RFC´, COF
do not account for polymorphism	MPC, ICP, NIH-ICP, IH-ICP, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC

Table 12: Mapping of measures to options for accounting for polymorphism

How to assign methods and attributes to classes

The final question is to decide to which class an attribute or method belongs. We have to decide, if inherited methods and attributes belong to the inheriting class or not. We distinguish two cases:

- When we compute the coupling of a class, we have to determine what are the methods/attributes of the class, and therefore contribute to the coupling of the class. The available options are:
  - only methods and attributes implemented in the class contribute to the coupling of the class
  - all methods and attributes implemented or declared in the class contribute to the coupling of the class
- When we count the frequency of connections according to option F) (i.e., for a given class, we count the number of other classes it is connected to, cf. criterion 3), we have to assign the items at the other ends of the connections to a class. The available options also depend on whether we are counting import or export coupling and are summarized in Table 13.

Import Coupling	Export Coupling
Assign server item to the class(es) where it is implemented (may be several classes if we account for polymorphism).	Assign client item to the class where it is implemented.
Assign server item to the class used to reference it.	---

Table 13: Options to assign methods and attributes to a class

For the measures defined in Section 4.3, only methods and attributes implemented in a class contribute to the coupling of the class.

5.2 Application of the framework

We apply the framework to select existing measures or to derive new measures for a given measurement goal. Application is performed by the following two steps:

- For each criterion of the framework, choose one or more of the available options basing each decision on the objective of measurement. The criteria must be dealt with in the order introduced in Section 5.1 because, as explained below, a decision made for one criterion can restrict the available options for subsequent criteria.
- Choose the existing measures accordingly or, if none exist to match the decisions made, construct new coupling measures. Remember that properties such as those presented in Section 4.3.9 can also be used to guide the definition and theoretical validation of new measures.

In the context of applying this framework, the measurement goal must at least specify

- The development phase at which measurement is to take place.
- The underlying hypothesis which drives measurement. The hypothesis will be of the form "Internal attribute coupling (as measured by the coupling measures defined) has a causal effect on external quality attribute Y." The external attribute Y could be maintainability, reliability, etc. As discussed in [BEM95], we believe that product measures by themselves, no matter how well defined, are not guaranteed to capture any relevant phenomenon regarding the quality of the system under study. It must be shown empirically that they are related to some external system quality attribute of interest. In other words, it is crucial to provide evidence that they are relevant quality indicators in order to be used and relied upon.

It is recommended to first define measures for the external attribute in the hypothesis and then apply the framework to derive coupling measures. Having an operational definition of the external quality attribute may help in the processes of choosing the appropriate coupling measures.

We now discuss, for each criterion, how the external attribute and the target development phase from a given measurement goal influences the choice of the available options and, where applicable, other aspects that may also impact this choice. We then illustrate the process of constructing an appropriate coupling measure by means of an example.

5.2.1 Type of connection

Influence of development phase

The selection of one or more types clearly will be influenced by the development phase at which the measure is aimed at. We are confined to types of connections that are applicable at the target development phase. In Table 4, we indicated for each type of connection from which development phase on it is applicable.

Influence of external attribute

It is difficult to provide guidelines for how a given external attribute affects the choice for one or more of the available types of connections. There are no obvious guidelines we could provide, and as of yet, only little practical experience has been gathered that we could report (see Section 6.0). Perhaps the exact definition of the external attribute gives some clue as to which types of connections will be relevant. To begin with, we recommend to choose several types of connections and conduct statistical analyses to investigate which types of connections are relevant, i.e., support empirically the underlying hypothesis of the measurement goal.

Additional remarks

Different types of connections have different strengths. For instance, according to information hiding principles, referencing an attribute of another class is worse (i.e., stronger) than invoking a method of another class. Therefore, we recommend not to incorporate more than one type of connection into a single measure. Rather, separate measures for each type of connection should be used. Exceptions to this rule may be justified, if the underlying hypothesis of the measurement goal does not require that different types of connections be differentiated, if the measure is coarse (cf. criterion 3, granularity). We will come back to this issue in the description of criterion 3.

5.2.2 Locus of impact

Influence of development phase

None.

Influence of external attribute

High import coupling of, e.g., a class indicates that the class depends strongly on other classes and their methods and attributes. Import coupling may therefore be relevant in conjunction with the following external attributes:

- Understandability: to understand a method or class, we must know about the services the class uses.
- Error-proneness: for similar reasons as understandability: if we incorrectly use an external service because we misunderstand it, we are likely to introduce errors.
- Maintainability: low understandability and high error-proneness result in low maintainability.
- Reusability: if a class depends on a large a large amount of external services, it will be more difficult to reuse it in other systems (because the external services will have to be made available too in the other systems).

High export coupling of, e.g., a class means that the class is used a lot by other classes and their methods and attributes. Export coupling may be relevant in conjunction with the following external attributes:

- Criticality: Any defects in a class with high export coupling are more likely to propagate to other parts of the system. Such defects are more difficult to isolate. In that respect, classes with high export coupling are particularly critical. An export coupling measure could therefore be used to select classes that should undergo special (effective but costly) verification or validation processes.

- Testability: A classes with high export coupling can be difficult to test. If defects need to propagate to other parts of the system to cause failures there, they may not be detected when testing the class in isolation.

5.2.3 Granularity

Domain of the measure

The finest possible domain of the measure has already been determined by criteria 1 and 2, and is evident from Table 4: it is either the client or server item of the chosen type of connection. For example, if we decide to measure import coupling and connections of type 5 (references from methods to attributes), the finest possible domain of the measure is the method (see row 5, column “client item”): for each method, count how often attributes are referenced by the method. Likewise, if we decide to measure export coupling and connections of type1 (aggregation), the finest possible domain of the measure is the class: for each class, how often is it used in other classes as type of an attribute.

Influence of development phase

The development phase influences the choice of domain indirectly only, in that it determines the finest possible domain of the measure. As explained above, the finest possible domain is determined by criteria 1 and 2. The target development phase is the main decisive factor for criterion 1. Thus, the target development phase has an indirect influence on the domain of the measure.

Influence of external attribute

Similar to the development phase, the external attribute has an indirect influence on the choice of domain. The finest possible domain is in part determined by criterion 2, for which the external attribute is the main decisive factor.

However, for some measurement goals we may want to choose a coarser domain for the measure than the finest possible. For instance, if the measurement goal is to characterize understandability, reusability etc. of a class, a measure defined at the class level is appropriate. This can be accomplished by taking into account all connections starting from or ending in the methods or attributes of the class. Similarly, measures defined on sets of classes or the whole system can be constructed by taking into account the connections starting from or ending in methods or attributes of the relevant classes. The ICP family of measures in Section 4.3 provides an example of how a measure defined on the method level (ICP(m)) is scaled up to the class level (ICP(c)) and set-of-classes level (ICP(SS)); see Table 2.

How to count connections

For this criterion, we have six options A) to F), where A) yields the finest measures and F) yields the coarsest measures; see Table 7.

Influence of development phase

The less precise and stable the information about the connections, the coarser the measure should be. If the information about the actual connections is detailed enough at the design phase we aim at, we may count individual connections. If the information we have is less detailed, or likely to change in the future, we may want to use a coarser measure. Generally speaking, the later in the design process the application of the measure, the more precise and stable the available information, the finer the measure.

Based on commonly used design methods, our recommendation is to use options D) to F) for analysis and high-level design, and options A) to D) for low-level design and implementation.

Influence of external attribute

In some cases, we may only need coarser grain measures. For example, assume we want to characterize understandability of a class for which the source code is available. The hypothesis is that the more methods invoked from other classes, the lower the understandability. To test this hypothesis, we count the methods used by the class according to option E), even though the source code is available: According to the hypothesis, the number of methods is decisive for understandability, not how often the methods are used.

Additional remarks

The coarser the measure, the less it says about the actual strength of connections between classes. Therefore, a single coarse measure might just as well take into account several types of connections at once: even if different types of connections have different strengths, mixing the types is acceptable for coarse measures (options E) and F)), because the strength of connections is not accounted for by the measure. In Section 4.3, CBO and COF are examples for coarse measures (option F)) which take more than one type of connection into account.

As was demonstrated in the theoretical validation of measures in Section 4.3.9, measures which count multiple connections between the same items as one do not fulfil property Coupling.5. Options B), D), E) and F) count the frequency of connections in this manner. Therefore, we must not use options B), D), E) or F) if we want our measures to fulfil property Coupling.5.

5.2.4 Stability of server

Influence of development phase

None.

Influence of external attribute

In the following, we list for each option some examples illustrating when it may be appropriate.

- Option I (count connections to unstable classes only) could be used for measuring reusability. Typically, library classes are easy to reuse. Import coupling from library classes may therefore be neglected.
- Option II (count connections to stable classes only) could be used for change impact analysis, where a libraries are to be replaced by other, possibly more efficient, libraries. Import coupling from the old libraries will be an indicator for the effort required to update a system to the new libraries.
- Option III (no distinction of stable and unstable classes) could be used when analyzing understandability. Hypothesis: understandability is influenced by the number of services used. It should not matter if the server classes are stable or not. (Additional assumption needed here: stable and unstable classes are equally well known).
- Option IV (count separately connections to both stable and unstable classes) could be used when analyzing maintainability. Hypothesis: maintainability is influenced by dependencies on both stable and unstable classes, and dependencies on unstable classes weigh heavier. To verify this hypothesis, coupling with stable and unstable classes has to be measured separately.

5.2.5 Direct or indirect connections

Influence of development phase

None.

Influence of external attribute

Indirect connections can be relevant when estimating the effort for run-time activities such as testing and debugging, or to estimate the impact of a modification to a class *c* on the system: the modification may necessitate other modifications to classes directly and indirectly connected to class *c* (ripple effects).

Indirect connections may also be relevant for reusability: if a class *c* is to be reused in another system, not only the classes to which *c* is coupled have to be provided in the system, but also classes required by these coupled classes and so on.

Direct connections are sufficient for the analysis of understandability: to understand a class, we need to know the functionality of the services directly used by the class. We do not need to know how these services are implemented, and therefore, what other services these services need (at least, this is true if the services are well documented).

5.2.6 Inheritance

There are three options for inheritance that must be considered.

Inheritance-based vs. non-inheritance-based coupling

Influence of development phase

None.

Influence of external attribute

It is difficult to provide guidelines here. There are no obvious guidelines, and as of yet, only little practical experience concerning the relative importance of inheritance-based and non-inheritance-based coupling is available: in Section 6.0, a study reported in [BDM96] is described, which showed that inheritance-based coupling is not a significant predictor for fault-prone classes, whereas non-inheritance-based coupling is.

To begin with, we recommend to measure both inheritance-based and non-inheritance-based coupling separately (option III), and conduct statistical analyses to investigate their relative importance.

Polymorphism

Influence of development phase

None.

Influence of external attribute

To analyze understandability, we do not need to account for polymorphism: the methods that are invoked through a method invocation have the same signature and should provide the same functionality (if we know one method, we know them all).

Accounting for polymorphism could be important for the analysis of error-proneness and testability. Because each method in *PIM*(*m*) has its own implementation, defects can propagate to or originate from any of these methods.

How to assign methods and attributes to classes

Influence of development phase

None.

Influence of external attribute

None.

Additional remarks

The choice for one of the available options is largely influenced by the decision made for the inheritance-based vs. non-inheritance-based coupling criterion: if we count inheritance-based coupling (options I, III or IV), “the attributes and methods of a class” have to be those implemented in the class. Because with inheritance-based coupling we measure the degree to which a class *c* is coupled to its ancestors, it makes no sense to assign the methods and attributes *c* has inherited to class *c*.

Assigning inherited methods and attributes to a class only makes sense when we analyze the coupling of a class *c* “as a whole” to other classes not related to *c* via inheritance. Connections from or to methods and attributes that class *c* inherits contribute to the coupling of class *c*. We cannot provide an example that shows the application of this option. However, this does not imply that there are no useful applications of this option. Therefore, we leave it as a part of the framework.

5.2.7 Construction of the coupling measures

After selection of one or more options for each criterion of the framework, we can construct a set of coupling measures accordingly. Let us assume that our measurement goal is to analyse the source-code of a system XYZ to predict maintenance effort where maintenance effort might be defined as the number of person hours spent fixing faults in a class or implementing changes to a class as a result of requirements changes. This data is measured over a certain period of time, say, one year. For each criterion of the framework, the following decisions are made.

- Type of connection (criterion 1): method invocations (option 6).
  Justification: At the source code level, assuming the system is a pure object-oriented implementation, method invocations are hypothesized to be the most relevant type of connection.
- Locus of impact (criterion 2): Count import coupling.
  Justification: If a method invokes many other methods, it is more likely to be affected by changes to the invoked methods.
- Granularity (criterion 3):
  - a) Required domain is “class”.
    Justification: Maintenance effort is defined at the class level.
  - b) Count individual connections (option C).
    Justification: The more often a method is invoked, the more effort is likely to be required to modify the invoking method when modification of the invoked method takes place.
- Stability of server (criterion 4): only count connections to unstable classes (option I).
  Justification: Assume that stable classes are reliable and will not need modification even as a result of requirements changes.
- Indirect or direct connections (criterion 5): Count both types of connections.
  Justification: there is no clear rationale for choosing one particular type of connection so all the available options should be investigated. For the purpose of illustration in Figure 5, however, only direct connections are considered.
- Inheritance (criterion 6):
  - a) Count both inheritance-based and non-inheritance-based coupling and distinguish between these types of coupling (option IV).
    Justification: We do not know whether inheritance-based or non-inheritance-based coupling is more important. Therefore, we need to measure both types of coupling separately to investigate their relative importance.
  - b) Account for polymorphism.
    Justification: Any method that is used by a class may give rise to a modification to the class. Therefore, we must include all methods that can be possibly invoked through polymorphism and dynamic binding.
  - c) Only methods implemented in a class contribute to the coupling of the class.
    Justification: We must choose this option because we count inheritance-based coupling (criterion 6a above). Furthermore, since we count individual connections (criterion 3b) we do not need to distinguish between declared and implemented methods of a class.

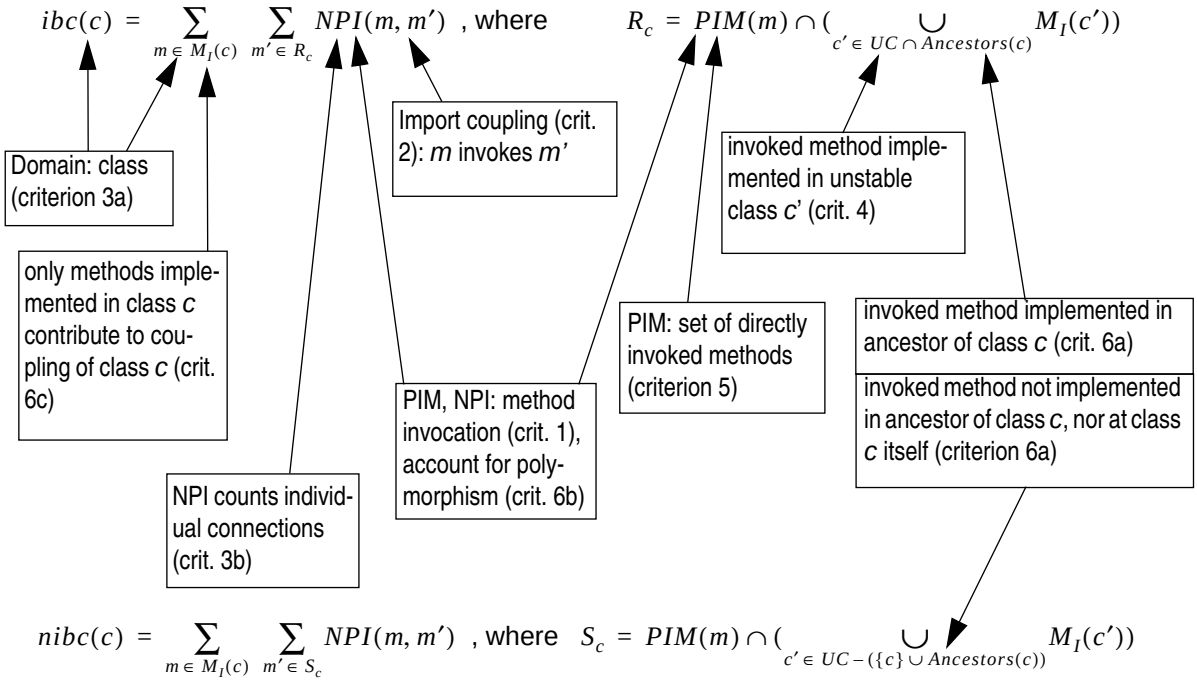


FIGURE 5. Example of how the criteria are reflected in a measures’ definitions

The corresponding coupling measures are shown in Figure 5. For measure “ibc” (inheritance-based coupling) we indicate for each criterion, where it is reflected in the definition of the measure. The only difference to measure “nibc” (non-inheritance-based coupling) is caused by criterion 6a) and is also indicated in Figure 5. For the distinction between stable and unstable classes, let us assume that we have a partition of the set *C* of all classes: $C = SC \cup UC$, where $SC \subseteq C$ is the set of stable classes in *C*, $UC \subseteq C$ is the set of unstable classes in *C*, and $SC \cap UC = \emptyset$.

5.2.8 Summary

We conclude the discussion of the unified framework with the following remarks.

- The measures generated with this framework are counts of connections between classes. This leads to the to the highest level of measurement, ratio measurement, which means the most powerful types of statistical analysis techniques can be performed.
- These measures, however, are not guaranteed to be useful. To be useful, the measures must be empirically validated with respect to the external quality attribute of interest specified in the measurement goal. We believe that measurement of internal product attributes are not meaningful in isolation, but only if they capture relevant external quality attributes; for full details, see [BEM95].
- Existing measures have been classified according to the options available for each criterion of the framework. This classification allows existing measures to be compared and their potential use identified. The classification has shown that some particular options of the framework criteria have no or only few corresponding measures proposed.
- Most object-oriented design methods define their own, specific types of connections between classes (e.g., “stimuli” in Jacobson’s OOSE method [JCJO92] and “links” in Rumbaugh’s OMT method [RBPEL91]). The presented framework may be tailored to a particular design method by adding the types of connections that are unique to the design method to the list of possible types of connections (criterion 1).

6.0 Empirical Validation Studies

In this section, we discuss empirical studies that have been performed with the reviewed coupling measures. In Section 6.1, we focus on studies conducted to empirically validate measures, i.e., show the usefulness of the measures. These are studies where

- the relationship of an internal attribute to an external quality attribute of a software product has been investigated, and
- at least one of the measures discussed in Section 4.3 has been used to measure the internal attribute.

We present the results of these studies in detail and analyse their validity. In Section 6.2, we give a brief overview of empirical work other than validation that has been performed using the reviewed coupling measures. Section 6.3 summarizes the results of this section.

6.1 Empirical validation studies

We are aware of only four publications that empirically validate some of the measures presented in Section 4.3. Table 14 provides a brief overview of this research.

As we can see, identical systems and dependent variables are used for the analyses in [LH93] and [LHKS95], and in [BBM96] and [BDM96]. In Section 6.1.1, we describe the systems and dependent variables used in the studies in more detail, in Section 6.1.2, we introduce measures used in the studies that have not already been discussed. In Section 6.1.3 we presents the results from the analyses, and in Section 6.1.4 we analyse the validity of these results.

Publication	Systems analyzed	Dependent Variable	Independent Variables
Li and Henry [LH93]	two medium-sized, commercial software systems	Maintenance effort (number of lines changed over a period of three years)	Chidamber and Kemerer's metrics suite [CK91] + MPC, DAC and two size measures (SIZE1, SIZE2)
Li *et al.* [LHKS95]			Chidamber and Kemerer's metrics suite [CK91] + MPC
Basili *et al.* [BBM96]	eight small software systems, developed in a students' project	Fault-proneness (probability of detecting a fault in a class)	Chidamber and Kemerer's metrics suite [CK94]
Briand *et al.* [BDM96]			Metrics suite defined in [BDM96] (FCAEC, OCAIC, ...)

Table 14: Overview of empirical validation studies

6.1.1 Systems and dependent variables used

The systems and dependent variables used in the studies are described in Table 15. We will refer to the systems and dependent variable used in [LH93, LHKS95] and [BBM96, BDM96] as "setting I" and "setting II", respectively.

	Setting I ([LH93] & [LHKS95])	Setting II ([BBM96] & [BDM96])
Systems	two commercial systems (UIMS, QUES)	eight systems from a student's project, developed by eight teams in four months
Application domain of systems	UIMS: User Interface System QUES: Quality Evaluation System	Information system for video rental businesses
Design method	Classic-Ada design language (object-oriented extension of Ada)	OMT (Rumbaugh *et al.* [RBPEL91]
Implementation language	Classic-Ada programming language	C++
Size	39 classes (UIMS), 70 classes (QUES)	total of 180 classes in all eight systems, sizes of systems between 5 and 14 KSLOC
Developers	- no information available -	eight groups of three students, had already some experience with C or C++, no previous experience with object-oriented analysis/design
Dependent Variable	Maintenance effort: number of lines changed in a class over a period of 3 years, calculated as follows: each deleted line counts 1, each added line counts 1, each modified line counts 2 (one deletion and one addition)	Fault-proneness: were faults detected in a class during acceptance testing (yes or no). Each system underwent eight hours of acceptance testing, detected faults were then traced back to classes.
Modeling technique	linear least-square regression	logistic regression

Table 15: Description of systems and dependent variables

6.1.2 Additional measures

The empirical validation studies also use measures that are not coupling measures and therefore have not been previously introduced. We now present the definitions of these measures.

In [LH93], two size measures SIZE1 and SIZE2 are used. SIZE1 of a class is defined as the number of non-inherited methods and non-inherited attributes of the class. Formally, we can define this measure as $Size1(c) = |A_I(c)| + |M_I(c)|$. SIZE2 of a class is defined as the number of semicolons in the class.

In [LH93], [LHKS95] and [BBM96], measure WMC from the metrics suite by Chidamber and Kemerer ([CK91] and CK94]) is used. This measure has originally been defined as follows [CK94]:

"Consider a class C_1, with methods $M_1, M_2, ..., M_n$. Let $c_1, c_2, ..., c_n$ be the complexity of the methods. Then:

$$WMC = \sum_{i=1}^{n} c_i \text{ ."}$$

The complexities c_i were intentionally left undefined. Two versions of WMC are used in the studies:

- In [LH93] and [LHKS95], c_i is defined as McCabe's cyclomatic complexity of method M_i [McC76]. We will denote this version of WMC as WMC-MCC. WMC-MCC only takes non-inherited methods of the class into account.
- In [BBM96], each c_i is set to one. In other words, this version of WMC counts the (non-inherited) methods of the class. We will denote this version as WMC-1.

In [LH93] and [LHKS95], a measure NOM (number of methods) is used, which too counts the number of non-inherited methods of a class and is thus identical to WMC-1. In the following, we will not use the name NOM, but refer to this measure as WMC-1 only.

In [LH93] and [LHKS95], the measure LCOM from [CK91] is used. This measure is defined as follows:

$$LCOM(c) = \left|\{\{m_1, m_2\} \middle| m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \emptyset\}\right|$$

It is the number of pairs of methods which do not use an attribute of *c* in common. In [BBM96] and [BDM96], the measure LCOM from [CK94] is used. We define this measure as follows:

Let $$P = \begin{cases} \emptyset, \ if \ AR(m) = \emptyset \ \forall m \in M_I(c) \\ \{\{m_1, m_2\} | m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) = \emptyset\}, \ else \end{cases} .$$

Let $$Q = \{\{m_1, m_2\} | m_1, m_2 \in M_I(c) \wedge m_1 \neq m_2 \wedge AR(m_1) \cap AR(m_2) \cap A_I(c) \neq \emptyset\} .$$

Then $$LCOM^*(c) = \begin{cases} 0, \ if \ |P| < |Q| \\ |P| - |Q|, \ else \end{cases} .$$

We denote this measure by *LCOM** to distinguish it from the '91 version of *LCOM* defined above.

In all studies described here, measures DIT (depth in inheritance tree) and NOC (number of children) from the metrics suite by Chidamber and Kemerer are used. The definitions in [CK91] and [CK94] are identical. DIT for a class is defined as the "maximum length from the node to the root of the tree". We give a recursive definition of DIT.

$$DIT(c) = \begin{cases} 0 \ , if \ Parents(c) = \emptyset \\ 1 + max\{DIT(c') | c' \in Parents(c)\} \ , otherwise \end{cases}$$

Measure NOC is straightforward to define: $NOC(c) = |Children(c)|$

6.1.3 Results of the studies

In [LH93] and [LHKS95], the metrics suite by Chidamber and Kemerer, SIZE1, SIZE2, MPC and DAC were tested. Li *et al.* proposed several multivariate regression models and conducted least-square regression and F-tests to estimate their predictive power.

In [BBM96] and [BDM96], the metrics suite by Chidamber and Kemerer and the suite of coupling measures introduced in [BDM96] were tested. Univariate logistic regression was performed to test the predictive power of each measure in isolation. Then, several multivariate models were built in a stepwise selection process, and tested.

An overview of the various multivariate regression models is given in Table 16. We assigned numbers to each model (MVM1 to MVM7) which we will use to reference them later.

Number	Source	Setting	Independent variables
MVM1	[LH93]	I	SIZE1, SIZE2, DIT, NOC, MPC, RFC, LCOM, DAT, WMC-MCC, WMC-1
MVM2	[LH93]	I	SIZE1, SIZE2
MVM3	[LH93]	I	DIT, NOC, MPC, RFC, LCOM, DAT, WMC-MCC, WMC-1
MVM4	[LHKS95]	I	DIT, RFC, LCOM, DAT, WMC-MCC, WMC-1
MVM5	[BBM96]	II	DIT, RFC, NOC, CBO, Class origin (0 if class has been reused verbatim, 1 otherwise)
MVM6	[BDM96]	II	DIT, RFC, OCMEC, FMMEC
MVM7	[BDM96]	II	DIT, WMC, OCAIC

Table 16: Overview of multivariate models used

Li *et al.* applied each model MVM1 to MVM4 to both their systems UIMS and QUES. They provide the R^2, adjusted R^2 and p-values from the F-tests for regression models MVM1 to MVM4, but do not provide the regression coefficients and p-values for each independent variable (except for MVM3, where the regression coefficients are given). In Table 17, we reproduce the results from the tests of MVM1 to MVM4. For models MVM5 to MVM7 by Basili *et al.* and Briand *et al.*, we have the regression coefficients and p-values for each independent variable, but not for the whole model. The results from these tests can be found in Table 18.

Number	Setting/ System	R^2	adjusted R^2	p-value
MVM1	I (UIMS)	0.9096	0.8773	0.0001
	I (QUES)	0.8737	0.8550	0.0001
MVM2	I (UIMS)	0.6617	0.6429	0.0001
	I (QUES)	0.6282	0.6172	0.0001
MVM3	I (UIMS)	0.9030	0.8871	0.0001
	I (QUES)	0.8680	0.8553	0.0001
MVM4	I (UIMS)	0.80	0.77	0.0001
	I (QUES)	0.69	0.67	0.0001

Table 17: Goodness of fit for multivariate models ([LH93], [LHKS95])

In Table 18, we summarize for each measure the results from the univariate analyses performed and the results from multivariate analyses MVM3, MVM5, MVM6 and MVM7. We aimed at organizing the table in such a manner that, for each measure, it is obvious which univariate and multivariate analyses it has been used in and the results of these analyses. The description of the columns is as follows:

- Column “Measure”: the name of the measure.
- Column “Model”: The prediction model. UV stands for univariate analysis; for multivariate analysis, we indicate the number of the respective multivariate regression model.
- Column “Setting: The setting in which the analysis took place. As explained in Table 15, in setting I the dependent variable is “number of lines changed”, and least-square regression has been used. For setting I, we also indicate in brackets which of the systems UIMS or QUES has been used in the respective analysis. In setting II, the dependent variable is “probability to detect a fault”, logistic regression is used, the systems analysed are the eight systems from students’ projects.
- Column “Coefficient”: The regression coefficient of the measure in the respective model.
- Column “$\Delta\psi$” An evaluation of the impact of the measure on the dependent variable for logistic regression. For univariate analyses in setting II, the value $\Delta\psi$, which is based in the notion of odds ratio, is provided. The odds ratio $\psi(X)$ represents the ratio between the probability of having a fault and not having a fault when the value of the measure is *X*. $\Delta\psi$ is defined as $\Delta\psi = \psi(X+1)/\psi(X)$, i.e., $\Delta\psi$ represents the reduction or increase in the odds ratio (expressed in the table as a percentage) when the value *X* increases by one unit.
- Column “p-value”: The statistical significance. In setting I, F-tests are used, in setting II a likelihood ratio test.

- Column “Sigfct.”: Indicates if the measure has been found significant. In setting I, no level of significance is given. In setting II, a threshold of $\alpha = 0.05$ is used.

For table cells which have been left empty, the corresponding information is either not available or not applicable.

Measure	Model	Setting/ System	Coefficient	$\Delta\psi$	p-value	Sigfct.
MPC	MVM3	I (UIMS)	-2.58682			
		I (QUES)	0.169582			
DAC	MVM3	I (UIMS)	12.92241			
		I (QUES)	-0.950382			
LCOM	MVM3	I (UIMS)	2.762436			
		I (QUES)	-2.195476			
WMC-MCC	MVM3	I (UIMS)	2.523366			
		I (QUES)	0.886097			
WMC-1	MVM3	I (UIMS)	-6.78521			
		I (QUES)	1.94425			
	UV	II	0.022	2%	0.0607	yes
	MVM7	II	0.10		0.0003	yes
DIT	MVM3	I (UIMS)	2.4950300			
		I (QUES)	-2.151131			
	UV	II	0.485	62%	0.0000	yes
	MVM5	II	0.5		0.0004	yes
	MVM6	II	1.15		0.0000	yes
	MVM7	II	0.86		0.0000	yes
NOC	MVM3	I (UIMS)	5.368791			
		I (QUES)	0.0			no
	UV	II	-3.3848	-96%	0.0000	yes
	MVM5	II	-2.01		0.0178	yes
RFC	MVM3	I(UIMS)	1.797583			
		I(QUES)	-0.14156			
	UV	II	0.085	9%	0.0000	yes
	MVM5	II	0.11		0.0000	yes
	MVM6	II	0.10		0.0000	yes
CBO	UV	II	0.142	15%	0.0000	yes
	MVM5	II	0.13		0.0072	yes
LCOM*	UV	II				no
OCAIC	UV	II	0.32	38%	0.0056	yes
	MVM7	II	0.1		0.0050	yes
IFCAIC	UV	II				no
ACAIC	UV	II				no
OCAEC	UV	II				no
FCAEC	UV	II				no
DCAEC	UV	II				no
OCMIC	UV	II	0.093	10%	0.0010	yes
IFCMIC	UV	II				no
ACMIC	UV	II				no

Table 18: Regression coefficients for individual measures

Measure	Model	Setting/ System	Coefficient	$\Delta\psi$	p-value	Sigfct.
OCMEC	UV	II	0.11	12%	0.0001	yes
	MVM6	II	0.15		0.0004	yes
FCMEC	UV	II				no
DCMEC	UV	II				no
OMMIC	UV	II	0.115	12%	0.0000	yes
IFMMIC	UV	II	0.485	62%	0.0000	yes
AMMIC	UV	II				no
OMMEC	UV	II	0.06	6%	0.0000	yes
FMMEC	UV	II	0.868	138%	0.0000	yes
	MVM6	II	0.28		0.0054	yes
DMMEC	UV	II				no

Table 18: Regression coefficients for individual measures

Discussion of the results

- In setting II, the measurement values of LCOM* showed small variance, so that this measure could not be used as predictor. Basili *et al.* attributed this to the definition of LCOM*: the measurement value is set to zero whenever there are more pairs of methods in a class which use an attribute in common than pairs of methods which do not. As a result, LCOM4 is zero for a large number of classes which otherwise would yield (different) negative values for LCOM*.
- In setting I, multivariate model MVM3, the regression coefficients for measures DIT, MPC, LCOM, RFC and MPC have opposite signs in systems UIMS and QUES. This is not explained by Li and Henry.
- Many measures of the suite defined in [BDM96] show only a small variance. The authors suggest that either this is a peculiarity of the data set they used, or the measures could be too coarse. In particular, we can observe that none of the measures for inheritance-based coupling was found to be significant.
- Measure NOC is zero for all classes in system QUES. This is not explained by Li and Henry.
- In setting II, the observed trend for NOC is contrary to what was expected: a large NOC indicates a lower probability of fault detection. Basili *et al.* explain this by the fact that classes that have been reused verbatim tend to have a higher NOC, and in a second study using the same systems, verbatim reused classes were shown to be less fault-prone [BBM95].

Comparison to size measures

In [LH93], [BBM96] and [BDM96], the authors investigate the question whether the measures they use are better predictors than pure size measures.

To that end, Li and Henry defined and tested the “size model” MVM2 in Table 16, using only variables SIZE1 and SIZE2. From the adjusted R^2‘s of 0.6429 and 0.6172 in UIMS and QUES, respectively, they concluded that the size model MVM2 is a predictor for maintenance effort (p-value 0.0001). Then they conducted a partial F-test to determine if there is a difference between the predictive power of MVM1 and MVM2. They rejected the hypothesis that there is no difference between MVM1 and MVM2 at a significance level of $\alpha = 0.005$, and concluded that MVM1 is a better predictor than MVM2.

Basili *et al.* [BBM96] calculate the correctness and completeness of MVM5: 60% of all classes would have been correctly predicted as faulty, and 88% of all faulty classes would have been detected. Then, they build a multivariate model of three size measures. With this model, 45.5% of all classes would have been correctly predicted as faulty, and 83% of all faulty classes would have been detected. The authors conclude that MVM 5 is a better predictor than the size-based multivariate model.

Briand *et al.* correlate the measures that have been found significant in univariate analysis to two size measures (number of source lines of code, number of executable statements). They find that some of their coupling measures are strongly related to the size measures, but that they still capture different dimensions (Spearman rank correlation coefficient r_s^2 is not approaching 1.0).

6.1.4 Threats to validity

We distinguish between three types of threats to validity of an empirical study:

- Construct validity: The degree to which the independent and dependent variables accurately measure the concepts they purport to measure.
- Internal validity: The degree to which conclusions can be drawn about the causal effect of the independent variables on the dependent variables.
- External validity: The degree to which the results of the research can be generalized to the population under study and other research setting.

We now apply these criteria to the studies described above.

Construct validity

The choice of the dependent variable used in [LH93] and [LHKS95] to measure maintainability (or maintenance effort) is questionable: The “number of lines changed in a class” may say little about the actual effort (in terms of person-hours etc.) spent on the maintenance of a class: some changes to a class may require more effort than others, even though they contribute the same or even less to the count of “number of lines changed”, and vice versa.

Internal validity

In [LHKS95], [BDM96], [BBM96] and, the function of some measures as “design measures” is emphasized, i.e., these measures are applicable in early stages of the development process. If these measures are found to be useful, they can be used to detect problems in the design before implementation starts, thus potentially saving time and effort for rework of the design. However, in the studies measurement was performed only after implementation. If measurement had taken place, say, before implementation started, different measurement data would have been obtained, because the description of the system before implementation is less complete than after implementation (for instance, the information which method invocations there are is less detailed before implementation than after implementation, method invocations are counted by CBO, RFC, MPC, OMMIC, IFMMIC, AMMIC, OMMEC, FMMEC, DMMEC). This in turn could have lead to completely different results in the statistical analyses. The validity of the measures in early development phases is therefore not shown. In order to demonstrate the usefulness of measures in early development phases, the measures must be applied to the deliverables of the early development phases.

External validity

Basili *et al.* list the following facts that may restrict the generalizability of their results [BBM96]. Consequently, these facts also apply to the results of [BDM96]:

- The systems developed are rather small: they lie between 5000 and 14000 source lines of C++ code.
- The systems developed have a limited conceptual complexity.
- The developers may not be as well trained and experienced as average professional programmers.

The first point (rather small systems) also applies to the studies in [LH93] and [LHKS95]: system UIMS has 39 classes, system QUES has 70 classes. We have no information concerning the conceptual complexity of the systems and the experience of the developers.

6.2 Other empirical studies

In this section, we briefly present other empirical work that has been done with the measures in Section 4.3.

In [AGE95] and [CK94], the measures proposed in these papers were applied to several systems. The distribution of the measures values was presented and an ad hoc interpretation of the data was done (e.g., explanations for differences in the distribution of the measures values of the various systems). These studies show that the measures are collectable in large systems. However, we cannot draw any conclusion about the usefulness of these measures from this discussion.

Sharble and Cohen [SC93] used the metrics suite by Chidamber and Kemerer [CK91] to compare two object-oriented design methods: a “data-driven” and a “responsibility-driven” design method. The data-driven method

emphasizes information modelling, the responsibility driven method stresses the services (“responsibilities” in the terminology of this design method) the system has to provide. They developed two versions of a system, one following the data-driven method, the other following the responsibility-driven method. The data-driven design yielded for each measure of Chidamber and Kemerer’s suite higher values than the responsibility-driven design. The authors concluded that the responsibility-driven design method produces less complex system designs.

6.3 Summary

Some of the measures presented in Section 4.3 have been found to be useful predictors of fault-proneness of a class and the numbers of lines changed in a class over a maintenance period. They also appear to be better predictors than simple, traditional size measures. As of yet, these results are only valid for relatively small systems and when based on stable design information once the system is fully implemented. The usefulness of the measures in large systems and/or for stable system design information before the system is implemented still remains to be tested.

The small amount of empirical validation work that has been published is of concern: more empirical work is required. One reason for the small amount of empirical validation work may be that measures are often provided without underlying empirical models; without these, there can be no hypothesis testing. Another reason is that for empirical validation we need to define and measure an internal and an external attribute, and measuring an external quality attribute can in practice be difficult. Measuring an internal attribute is relatively easy: measures are quickly defined and, in order to apply the measure, we only need access to the artifacts of some already existing systems. Typically, we have some systems available and can perform the measurement. As shown in Section 6.2, this kind of measurement is frequently reported. Measuring an external quality attribute is more problematic, because we need additional information besides the measured system. For instance, if we want to measure maintenance effort, we need a system for which we have the appropriate maintenance effort data. Typically, we do not have systems for which the additional required information is available, since few organizations have adequate measurement programs in place.

Trade-offs in the definition of the external attribute to circumvent these problems limit the usefulness of the study. For instance, in [LH93] and [LHKS95], maintenance effort has been defined as the number of lines changed in a class. One advantage is that the number of lines changed in a class can be calculated if we have several versions of the same system, i.e., we do not need actual effort data. However, as already highlighted, the number of lines changed is not a straightforward indicator of the actual effort spent on the maintenance of the class.

7.0 Conclusions

Based on a standardized terminology and formalism, we have provided a framework for the comparison, evaluation, and definition of coupling measures in object-oriented systems. This framework is intended to be exhaustive and integrates new ideas with existing measurement frameworks in the literature. Thus, detailed guidance is provided so that coupling measures may be defined in a consistent and operational way and existing measures may be selected based on explicit criteria.

We have also used this framework to review the state-of-the-art, about which we draw the following conclusions:

- There is a very rich body of ideas regarding the way to address coupling measurement in object-oriented systems.
- However, many measures are not based on explicit empirical models and, therefore, their intended application is a priori difficult to determine.
- Very few measures have undertaken a thorough empirical validation. In other words, the usefulness of many of the measures is currently not empirically supported.
- When empirical validations do exist, they are sometimes seriously flawed because of serious threats to the validity of their results, e.g., construct validity when the dependent variable used in the analysis does not capture accurately the external quality attribute of interest.

Therefore, it appears that, although many good ideas have been reported, there is too little empirical work in (coupling) measurement, especially in the context of object-oriented systems. This can only hinder effective research and the design of satisfactory solutions for the practitioners of measurement.

Future work includes the investigation of cohesion and inheritance complexity measurement for object-oriented systems. In addition, we plan to perform an exhaustive empirical investigation of the measures reported in this article.

Acknowledgments

The research reported here has been conducted with the framework of Jürgen Wüst's master's thesis on quality measures for object-oriented systems.

References

[AGE95] F. Abreu, M. Goulão, R. Esteves, "Toward the Design Quality Evaluation of Object-Oriented Software Systems", *5th International Conference on Software Quality*, Austin, Texas, USA, October 1995.

[BBM95] V. Basili, L. Briand, W. Melo, "Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented systems", *Technical Report, University of Maryland, Department of Computer Science*, CS-TR-3395, January 1995.

[BBM96] V.R. Basili, L.C. Briand, W.L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22 (10), 751-761, 1996.

[BDM96] L. Briand, P. Devanbu, W. Melo, "An Investigation into Coupling Measures for C++", *Technical Report ISERN 96-08, IEEE ICSE '97*, Boston, USA, (to be published) May 1997.

[BEM95] L. Briand, K. El Emam, S. Morasca, "Theoretical and Empirical Validation of Software Product Measures", *Technical Report, Centre de Recherche Informatique de Montréal*, 1995.

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

[BMB94] L. Briand, S. Morasca, V. Basili, "Defining and Validating High-Level Design Metrics", T*echnical Report, University of Maryland, CS-TR 3301, 1994.*

[BMB96] L. Briand, S. Morasca, V. Basili, "Property-Based Software Engineering Measurement", *IEEE Transactions of Software Engineering*, 22 (1), 68-86, 1996.

[CK91] S.R. Chidamber, C.F. Kemerer, "Towards a Metrics Suite for Object Oriented design", in A. Paepcke, (ed.) *Proc. Conference on Object-Oriented Programming: Systems, Languages and Applications* (OOPSLA'91), October 1991. Published in SIGPLAN Notices, 26 (11), 197-211, 1991.

[CK94] S.R. Chidamber, C.F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 20 (6), 476-493, 1994.

[CS95a] N.I. Churcher, M.J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design'", *IEEE Transactions on Software Engineering*, 21 (3), 263-265, 1995.

[CS95b] N.I. Churcher, M.J. Shepperd, "Towards a Conceptual Framework for Object Oriented Software Metrics", *Software Engineering Notes*, 20 (2), 69-76, 1995.

[CY91a] P. Coad, E. Yourdon, "Object-Oriented Analysis", *Prentice Hall*, second edition, 1991.

[CY91b] P. Coad, E. Yourdon, "Object-Oriented Design", *Prentice Hall*, first edition, 1991.

[EKS94] J. Eder, G. Kappel, M. Schrefl, "Coupling and Cohesion in Object-Oriented Systems", *Technical Report*, University of Klagenfurt, 1994.

[Fen91] N. Fenton, "Software Metrics: A Rigorous Approach", *Chapman and Hall*, 1991.

[HM95] M. Hitz, B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented systems", in *Proc. Int. Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[HM96] M. Hitz, B. Montazeri, "Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective", *IEEE Transactions on Software Engineering*, 22 (4), 276-270, 1996.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach", *ACM Press/Addison-Wesley*, Reading, MA, 1992.

[LH93] W. Li, S. Henry, “Object-Oriented Metrics that Predict Maintainability”, *J. Systems and Software*, 23 (2), 111-122, 1993.

[LHKS95] W. Li, S. Henry, D. Kafura, R. Schulman, “Measuring object-oriented design”, J*ournal of Object-Oriented Programming*, 8 (4), 48-55, 1995.

[LLWW95] Y.-S. Lee, B.-S. Liang, S.-F. Wu, F.-J. Wang, “Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow”, in *Proc. International Conference on Software Quality*, Maribor, Slovenia, 1995.

[Mar94] R. Martin, “OO Design Quality Metrics - An Analysis of Dependencies”, *Position Paper, Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA’94*, October 1994.

[McC76] T.J. McCabe, “A complexity measure”, *IEEE Transactions on Software Engineering*, 2 (4), 308-320, 1976.

[Mye78] G. Myers, “Composite/Structured Design”, *Van Nostrand Reinhold*, 1978.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, “Object-Oriented Modeling and Design”, *Prentice Hall*, 1991.

[SB91] R.W. Selby, V.R. Basili, “Analyzing Error-prone Systems Structure”, *IEEE Transactions on Software Engineering*, 17 (2), 141-152, 1991.

[SC93] R.C. Sharble, S.S. Cohen, “The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods”, *Software Engineering Notes*, 18 (2), 60-73, 1993.

[SMC74] W. Stevens, G. Myers, L. Constantine, “Structured Design”, *IBM Systems Journal*, 13 (2), 115-139, 1974.

[TZ81] P.A. Troy, S.H. Zweben, “Measuring the Quality of Structured Designs”, *J. Systems and Software* 2, 113-120, 1981.

Appendix

A Glossary of Terms

Standard terminology used for object-oriented concepts is provided as follows:

- Component: Any system entity whose properties may be measured. Most important are typically classes, methods, and attributes.
- Class: Compound structure encapsulating data and functional elements.
- Object: Instance of a class
- Attribute: A data item encapsulated in a class. Other names: instance variable, data member, state variable
- Method: a procedure or function encapsulated in a class. Other names: operation, service, member function
- Inheritance: “Is-a” relationship between two classes. A class c may inherit from class d. The methods and attributes of class d are then available to class c.
- Parent class: If class c inherits from class d, class d is a parent class. Other name: superclass.
- Child class: If class c inherits from class d, class c is a child class. Other name: subclass.
- Descendent class: The descendent classes of a class c are the children classes of class c, their children etc. (any class that directly or indirectly inherits from class c).
- Ancestor class: The ancestor classes of a class c are the parent classes of class c, their parent classes etc. (any class, from which c directly or indirectly inherits).
- Signature: Unique identifier that identifies a method. The signature specifies the method’s name, the parameters it takes, and the return type.
- Interface: The set of all signatures defined as public within a class, i.e., the interface characterises the complete set of messages that can be sent to an instance of that class.
- Body: The body of a method is its implementation. The body of a class is the implementation of its methods.
- Access Method: a method whose sole purpose is to provides access to one or more attributes of the class.
- Constructor: a method which creates and initializes an object of a class.
- Virtual method: A method which has no implementation. The implementation of the method is deferred to children classes.
- Abstract class: A class which has at least one virtual method. No objects can be instantiated from an abstract class.
- Message: Classes, via their methods, send messages to request services from other classes, possibly including specific recipients and parameters.
- Polymorphism: An identifier may refer to instances of different classes (typically having a common ancestor) at run-time, allowing objects to be bound to this identifier to respond to the same set of messages in different ways (however, the semantics of the response should be similar for all objects).

Also defined is the applicable measurement terminology. The definition of the terms “measure”, “internal attribute”, “external attribute”, “theoretical validation” and “empirical validation” are taken from [BEM95]:

- Measure, domain, range: Let *D* be a set of empirical objects to be measured (e.g., a set of classes, methods), and *R* be a set of formal objects (e.g., real numbers). A measure is a mapping $\mu: D \rightarrow R$, which maps every element of *D* onto an element of *R*. We call *D* the *domain* of measure μ, *R* the *range* of μ. Other name: metric.
- Internal attribute: A quality or property of a software product that can be measured in terms of the product itself, e.g., size, coupling, etc.
- External attribute: A quality or property of a software product that can not be measured solely in terms of the product itself. For instance, to measure maintainability of a product, measurement of maintenance activities on the product will be required in addition to measurement of the product itself.
- Operationally defined: A measure is considered operationally defined if no further interpretation of its definition is required to use it, i.e., it is stated in an unambiguous manner.
- Theoretical validation: A demonstration that a measure is really measuring the internal or external attribute it purports to measure.

- Empirical validation: A demonstration that a measure is useful in the sense that it is related to an interesting external attribute in an expected way.
- Measurement goal: Specification of the objectives of measurement in a given context. In this paper, it is assumed that the objective of measurement is to test a hypothesis of the form: “Internal attribute X has an impact on external attribute Y”, i.e., to conduct an empirical validation. Other information that typically is included in the measurement goal: the development phase at which measurement is to take place, the environment in which measurement is to take place (company, development team, methodology used etc.), properties for measures of internal or external attributes.

B Explaining the Formalism

This appendix provides illustrating examples for some of the definitions used in the formalism provided in Section 3.0.

Declared and implemented methods

In Definition 3, set $M_D(c)$ of methods *declared in c* and set $M_I(c)$ of methods *implemented in c* were defined.

Consider the example classes in Figure 6. For class *c, M(c)={c::m1, c::m2, c::m3}* where $M_D(c)$*={c::m1}* and $M_I(c)$*={c::m2, c::m3}*. For class *d*, because of inheritance, *M(d)={d::m1, c::m2, d::m3, d::m4}* with $M_D(d)$*={c::m2}* and $M_I(d)$*={d::m1, d::m3, d::m4}*.

```
class c {
  public:
    virtual void m1()=0;
    void m2(int);
    void m3(char);
};
void c::m2(int i) {/*...*/}
void c::m3(char ch) {/*...*/}
```

```
class d : public c {
  public:
    void m1();
    void m3(char);
    void m4();
};
void d::m1() {/*...*/}
void d::m3(char ch) {/*...*/}
void d::m4() {/*...*/}
```

FIGURE 6. Example methods

Notice that unique labels, e.g., “*c::m1*”, are used to represent any given method. In the case where a class inherits a method and does not override it, the method retains the identity provided by the parent class. In the example, class *d* inherits method *c::m2* of class *c* and does not override it. Thus, $M_I(c) \cap M_D(d) \neq \emptyset$ because $c::m2 \in M_I(c)$ and $c::m2 \in M_D(d)$. In the other cases each method is provided with a new identity. For example, $d::m3, d::m4 \notin M(c)$. In addition, certain object-oriented programming languages allow “method overloading” where a class can have two or more methods with the same name but different signatures. Such methods will still be provided with a unique identity.

Inherited, overriding and new methods

In Definition 4, sets $M_{INH}(c)$, $M_{OVR}(c)$ and $M_{NEW}(c)$ of inherited, overriding and new methods of a class *c* where introduced. In the example in Figure 6 it is $M_{NEW}(d)$={*d::m4*}, $M_{OVR}(d)$={*d::m1*, *d::m3*} and $M_{INH}(d)$={*c::m2*}.

For any class $c \in C$ and method $m \in M_{NEW}(c)$ there is no method with the same signature declared in any ancestor class of *c*. $M_{INH}(c)$, $M_{OVR}(c)$ and $M_{NEW}(c)$ form a partition of *M(c)*: they are pairwise disjoint, and $M_{INH}(c) \cup M_{OVR}(c) \cup M_{NEW}(c) = M(c)$.

Method invocations

Consider the example in Figure 7.We make the following remarks:

- Method *c::mc1* invokes method *d::md* using a pointer to an object of type *d*. Class *d* is the “static type” of the object being pointed to. At run-time, the dynamic type of the object may be of class *d* or any descendent class

of *d*. Due to polymorphism, the actual method being executed can be implemented in class *d* or any descendent class of *d* (in the example, either *d::md* or *d1::md* may be executed; *d2::md* is identical to *d::md*).

- The body of *c::mc1* contains a total of three statements which invoke method *d::md*. How often the method *d::md* is actually invoked at run-time, however, cannot be determined from static analysis.

```
class c {
     d *d_ptr;
     d2 d_obj;
  public:
     void mc1(int);
}
void c::mc1(int j) {
  d_ptr->md(-1);
  for( int i=1; i<=j; i++)
     d_ptr->md(i);
  d_obj.md()
}
```

```
class d {
  public:
     virtual void md(int);
};
void d::md(int i) {/*...*/}

class d1 : public d {
  public:
     void md(int);
};
void d1::md(int i) {/*...*/}

class d2 : public d {
  public:
     void md2();
};
```

FIGURE 7. Example method invocation

In the following, we show the values for *SIM, NSI, PIM* and *NPI* from Definitions 7 to 10 for method *c::m1* in Figure 7. It is *SIM(c::mc1)={d::md}* and *NSI(c::mc1, d::md)=3*. As a result of inheritance, the same method can be statically invoked for objects of different classes. Method *d::md* is statically invoked twice for an object of type *d*, and once for an object of type *d2*. It is *PIM(c::mc1)={d::md, d1::md}*, where *NPI(c::mc1,d::md)=3* and *NPI(c::mc1, d1::md)=2*. As a result of polymorphism, one method invocation can contribute to the *NPI* count of several methods.

Notice that the static type of an object for which a method *m´* is invoked may be an abstract class and *m´* may be a virtual method. The dynamic type of an object cannot be an abstract class because an object cannot be an instance of an abstract class.

Declared and implemented attributes

To illustrate declared and implemented attributes (Definition 11), consider the example in Figure 8, it is $A_D(c) = \emptyset$ and $A_I(c)=A(c)=\{c::x, c::y, c::z\}$. For class *d* $A_I(d)=\{d::a, d::b\}$, $A_D(d)=\{c::x, c::y, c::z\}$, and $A(d)=\{d::a, d::b, c::x, c::y, c::z\}$.

```
class c {
  public:
     int x,y,z;
     /* ... */
};
```

```
class d : public c {
     int a;
     const int b=2;
  public:
     int f();
  /* ... */
};
int d::f() {
  a=a+1; x=a; y=b; return a+b;
}
```

FIGURE 8. Example class with attributes

Like methods, attributes are modelled as elements of sets and require a unique identity. As with methods, non-inherited attributes of a class are provided with their own identity, whereas inherited attributes are provided with their identity in some ancestor class. Notice no distinction is made between constant attributes and regular attributes. For instance, for class *d* $d::b \in A(d)$.

Attribute references

In Definition 13, set *AR* was introduced to model the set of attributes referenced by a method. In Figure 8, it is *AR(d::f)={d::a, c::x, c::y, d::b}*. Notice there is no distinction between read and write accesses to attributes as none of the measures discussed make this distinction.

C A Generic Object-Oriented Development Process

In the survey of measures in Section 4.3, we classify the measures according to when during the development process they become applicable. To be able to classify the measures consistently, it must be known when certain deliverables required for measurement are available. To achieve this requirement, a generic development process with four steps and the deliverables available at the end of each is defined. These deliverables comprise of modelling concepts which are similar to those used by most object-oriented methodologies and are exemplified by means of a mapping to Jacobson's OOSE method [JCJO92]. In general, each step will be performed in several iterative cycles, the deliverables being updated as the problem and solution are more clearly defined.

- Analysis (An): The following deliverables are available at the end of the analysis phase:
  - High level classes: High level classes model the entities in the problem domain. A high-level class (HLC) will in later phases be implemented by one or more regular classes, i.e., classes as they are provided by programming languages. At the analysis phase we know nothing about the internal structure of HLCs. We do have a good idea of the services the HLC provides.
  - Inheritance relationships: we have knowledge of some inheritance relationships between HLCs, derived mainly as identification of "type-of" relationships. In general, the number of inheritance relationships identified during analysis will be relatively small.
  - Other relationships: These are relationships between HLCs such as "uses", "consists-of", etc. These relationships are derived on the basis of the services a HLC is to provide. For example, if HLC *A* requests a service which is provided by HLC *B*, there is a uses relationship between *A* and *B*.

  Note that it is also usual for the system to be decomposed into subsystems, i.e., groups of closely related HLCs. This occurs for ease of understandability and iterative enhancement.

  Mapping to OOSE: In the following, terms specific to the OOSE terminology are set in quotes to distinguish them from out standard terminology. The analysis phase corresponds to the "Robustness analysis" in the OOSE method. The artifacts described above are those found in the "Analysis model": HLCs are "objects" ("interface, entity or control objects"); inheritance relationships have their direct counterpart in OOSE; the other relationships are called "associations" ("communication associations, acquaintance associations"). The services provided by each HLC (i.e., "object" in OOSE) are not part of the "Analysis model", but are evident from the "use cases" defined in an earlier process step of OOSE.

  Most object-oriented methodologies feature an early analysis phase and introduce a graphical notation, where high-level classes are represented by boxes (or circles), and relationships (inheritance, uses, etc.) are represented by different kinds of arrows between circles or boxes). The information contained in these diagrams is considered to be the measurable output of the analysis.

- High-level design (HLD): During high-level design, the HLCs are refined. This involves the following decisions: which regular classes are needed to implement a high-level class, what methods must a HLC provide, what input and output parameters will the methods need, in which "regular" class should each method be implemented, what data will each class hold. Also, we will know which functionality each method has to fulfill, and have a rough idea about which other methods a method uses. The methods at this level are "high-level" methods. Several methods may be required later to implement one "high-level" method, that is, new methods will be added later. Also, the input and output parameters are still subject to later refinement.
  As the refinement of the HLCs creates new classes, new inheritance relationships between classes will arise. For instance, if we identified a number of classes which perform network communication, these classes are likely to have some functionality (methods) in common (e.g., wait for message, send message, receive message). This functionality could be factored out in a common parent class of the network communication classes.

  Mapping to OOSE: The high-level design corresponds to the first half of the "Construction" phase of the OOSE method. The HLCs are mapped onto "blocks", and each block consists of one or more classes (the implementation environment will influence the choice of classes). Using "interaction diagrams", "stimuli" be-

tween blocks are determined, and what information is passed with each “stimulus”. The “stimuli” correspond to the “high-level” methods, the information passed corresponds to the parameters.

- Low-level design (LLD): During low-level design, algorithms for each method are designed. Typically, techniques such as state-transition graphs, flowcharts, or program description languages (PDL) are used. The design of algorithms, as well as determining the precise signature for each method, is likely to identify the need for new methods and attributes. There is also detailed information about which methods and attributes are used by any given method.
  Further possibilities for class abstraction can still be discovered at LLD and the use of library classes is considered. This can result in some new classes being added to the system and minor rearrangement of the inheritance hierarchy.

  Mapping to OOSE: The low-level design phase corresponds to the second half of the “Construction” phase in OOSE. State-transition graphs are used to design algorithms for the methods of each class.
- Implementation (Imp): After implementation the source code is available.
  Mapping to OOSE: OOSE has a process step “Implementation” which produces the source code.

D Levels Of Coupling As Defined By Myers

In this appendix, we restate the definition of coupling given by Myers [Mye78] in the context of structured design. In the following, the term “module” refers to a set of one or more contiguous program statements having a name by which other parts of the system can invoke it. Myers identified six levels of coupling between modules. Listed from tightest (worst) to lowest (best) coupling, these are:

- *Content*: One module *directly* references the inside of the other, or the normal linkage conventions between the modules are bypassed. Almost any change to one module will require the other module to be changed, too.
- *Common*: Two or more modules reference a global data structure (named after the FORTRAN COMMON statement).
- *External*: Two or more modules reference a homogeneous global data item.
- *Control*: One module explicitly controls the logic of the other, i.e., it passes an explicit element of control (function codes, control-switch arguments, module-name arguments) to the other module.
- *Stamp*: Two modules reference the same nonglobal data structure (e.g., passing of an argument).
- *Data*: Two modules directly communicate with one another (e.g., one module calls the other module), and all interface data (e.g., input and output arguments) between the modules are homogeneous data items.