Dynamic Coupling Measurement for Object-Oriented Software

Erik Arisholm, *Member*, *IEEE*, Lionel C. Briand, *Member*, *IEEE*, and Audun Føyen

Abstract—The relationships between coupling and external quality factors of object-oriented software have been studied extensively for the past few years. For example, several studies have identified clear empirical relationships between class-level coupling and class fault-proneness. A common way to define and measure coupling is through structural properties and static code analysis. However, because of polymorphism, dynamic binding, and the common presence of unused ("dead") code in commercial software, the resulting coupling measures are imprecise as they do not perfectly reflect the actual coupling taking place among classes at runtime. For example, when using static analysis to measure coupling, it is difficult and sometimes impossible to determine what actual methods can be invoked from a client class if those methods are overridden in the subclasses of the server classes. Coupling measurement has traditionally been performed using static code analysis, because most of the existing work was done on nonobject oriented code and because dynamic code analysis is more expensive and complex to perform. For modern software systems, however, this focus on static analysis can be problematic because although dynamic binding existed before the advent of object-orientation, its usage has increased significantly in the last decade. This paper describes how coupling can be defined and precisely measured based on dynamic analysis of systems. We refer to this type of coupling as *dynamic* coupling. An empirical evaluation of the proposed dynamic coupling measures is reported in which we study the relationship of these measures with the change proneness of classes. Data from maintenance releases of a large Java system are used for this purpose. Preliminary results suggest that some dynamic coupling measures are significant indicators of change proneness and that they complement existing coupling measures based on static analysis.

Index Terms—Coupling measurement, change predictions, quality modeling, maintenance.

1 Introduction

In the context of object-oriented systems, research related to quality models has focused mainly on defining structural metrics (e.g., capturing class coupling) and investigating their relationships with external quality attributes (e.g., class fault-proneness) [7]. The ultimate goal is to develop predictive models that may be used to support decision making, e.g., decide which classes should undergo more intensive verification and validation. Regardless of the structural attribute considered, most metrics have been so far defined and collected based on a static analysis of the design or code [7], [10], [12], [13], [15], [16]. They have, on a number of occasions, proven to be accurate predictors of external quality attributes, such as fault-proneness [7], ripple effects after changes [11], [14], and changeability [1], [14]. However, many of the systems that have been studied showed little inheritance and, as a result, limited use of polymorphism and dynamic binding [17].

As the use of object-oriented design and programming matures in industry, we observe that inheritance and polymorphism are used more frequently to improve internal reuse in a system and facilitate maintenance. Though no formal survey exists on this matter, this is visible when analyzing the increasing number of open source projects, application frameworks, and libraries. The problem is that the static, coupling measures that represent the core indicators of most reported quality models [7] lose precision as more intensive use of inheritance and dynamic binding occurs. This is expected to result in poorer predictive accuracy of the quality models that utilize static coupling measurement.

Let us take an example, as illustrated in Fig. 1, to clarify the issue at hand. Due to inheritance, the class of the object sending or receiving a message may be different from the class implementing the corresponding method. For example, let object `a` be an instance of class `A`, which is inherited from ancestor `A'`. Let `A'` implement the method `mA'`. Let object `b` be an instance of class `B`, which is inherited from ancestor `B'`. Let `B'` implement the method `mB'`. If object `a` sends the message `mB'` to object `b`, the message may have been sent from the method source `mA'` implemented in class `A'` and processed by a method target `mB'` implemented in class `B'`. Thus, in this example, message passing caused two types of coupling: 1) object-level coupling between class `A` and class `B` (i.e., coupling between instances of `A` and `B`) and 2) class-level coupling between class `A'` and `B'`. The code may very well show statements where an object of type `A` invokes from `mA'` method `mB'` on an object of type `B`. However, to assume, through static code analysis, that there is class-level coupling between `A` and `B` as a result, is simply inaccurate. Both types of coupling, at the class and object levels, need to

- E. Arisholm and A. Føyen are with the Department of Software Engineering, Simula Research Laboratory, Lysaker, Norway. E-mail: erika@simula.no, audunf@ifi.uio.no.
- L.C. Briand is with the Software Quality Engineering Laboratory, Computer and Systems Engineering, Carleton University, Ottawa, Canada. E-mail: briand@sce.carleton.ca.

Manuscript received 12 Aug. 2003; revised 21 Jan. 2004; accepted 6 Apr. 2004.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0109-0803.

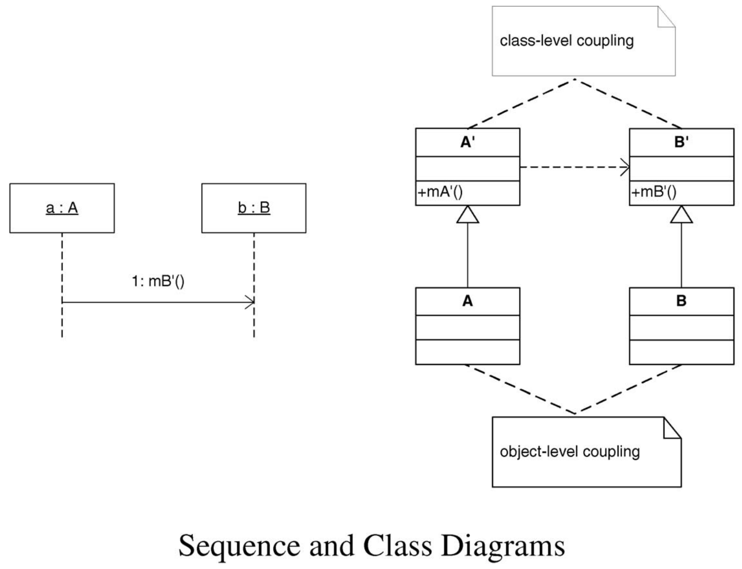

Fig. 1. Class-level versus object-level coupling.

be captured accurately to address certain applications and must be investigated.

We propose a set of coupling measures (referred to as *dynamic* coupling measures) that is defined on an analysis of runtime object interactions. They can be collected through a dynamic analysis of the code, that is, by executing the code and saving information regarding the messages that are being sent among objects at runtime. It is also, a priori, conceivable that dynamic design models (e.g., interaction diagrams in the Unified Modeling Language (UML) [5]) could be used to collect such measures.

Existing evidence suggests that dynamic coupling could be of strong interest. A preliminary empirical study on a SmallTalk system suggests that there is a significant relationship between change proneness and dynamic coupling [1], [2]. Furthermore, according to the results of a controlled experiment [3], static coupling measures may sometimes be inadequate when attempting to explain differences in changeability (e.g., change effort) for object-oriented designs. A follow-up study indicates that the actual flow of messages taking place between objects at runtime is often traced systematically by professional developers when attempting to understand object-oriented software [6]. The results thus suggest that dynamic coupling measures could be of interest as predictors of the cognitive complexity of object-oriented software. Finally, dynamic coupling is more precise than static coupling for systems with dead (unused) code, which is uninteresting in most situations and can seriously bias analysis.

This paper has two main objectives. First, it formally defines a set of dynamic coupling measures. Some of them can be measured in the context of object-oriented designs whereas others require the dynamic analysis of code. Second, it validates the measures in two distinct ways: 1) Their mathematical properties are systematically analyzed and 2) The statistical and practical significance of using dynamic coupling measures is empirically assessed in the context of models predicting the change proneness of Java components.

The remainder of this paper is organized as follows: Section 2 describes 12 dynamic coupling measures and highlights the ways in which they differ from static measures. These dynamic coupling measures differ in terms of the entities they measure and their scope and granularity, and are classified accordingly. They are defined in an informal, intuitive manner but also using a formal framework based on set theory and first-order logic. The main reason for the latter is to ensure that the definitions are precise and unambiguous to allow precise discussions of the measurement properties and the replication of empirical studies. Section 3 describes how the dynamic coupling measures can be collected. Section 4 presents a case study as a first empirical evaluation of the proposed dynamic coupling measures. Section 5 describes related research. Section 6 concludes and outlines future research.

2 Dynamic Coupling Measurement

We first distinguish different types of dynamic coupling measures. Then, based on this classification, we provide both informal and formal definitions, using a working example to illustrate the fundamental principles. Using a published axiomatic framework [10], we then discuss the mathematical properties of the measures we propose. Our measures were designed to fulfill five properties that we deem very important for any coupling measure to be well formed. In order to define measures in a way that is programming language independent, we refer to a generic data model defined with a UML class diagram.

2.1 Classifying Coupling Measures

There are different ways to define dynamic coupling, all of which can be justified, depending on the application context where such measures are to be used. Three decision criteria are used to define and classify dynamic coupling measures.

1. *Entity of measurement*. Since dynamic coupling is based on dynamic code analysis, coupling may be

TABLE 1
Dynamic Coupling Classification

Entity	Granularity (Aggregation Level)	Scope (Include/Exclude)
Object	Object Class (set of) Scenario(s) (set of) Use case(s) System	Library objects Framework objects Exceptional use cases
Class	Class Inheritance Hierarchy (set of) Subsystem(s) System	Library classes Framework classes

measured for a class or one of its instances. The *entity of measurement* may therefore be a class or an object.

2. *Granularity*. Orthogonal to the entity of measurement, dynamic coupling measurement can be aggregated at different levels of *granularity*. With respect to dynamic *object* coupling, measurement can be performed at the object level, but can also be aggregated at the class level, i.e., the dynamic coupling of all instances of a class is aggregated. In practice, even when measuring object coupling, the lowest level of granularity is likely to be the class, as it is difficult to imagine how the coupling measurement of objects could be used. Alternatively, all the dynamic coupling of objects involved in an execution scenario can be aggregated. We can also measure the dynamic object coupling in entire use cases (i.e., sets of scenarios), sets of use cases, or even an entire system (all objects of all use cases). In the case where the entity of measurement is a class, the aggregation scale is different as we can aggregate dynamic *class* coupling across an inheritance hierarchy, a subsystem, a set of subsystems, or an entire system. The relationships between various levels of granularity are formally described in Section 2.2.
3. *Scope*. Another important source of variation in the way we can measure dynamic coupling is the *scope* of measurement. This determines which objects or classes, depending on the entity of measurement, are to be accounted for when measuring dynamic coupling. For example, we may want, depending on the application context, to exclude library and framework classes.

At the object level, we may want to exclude certain use cases modeling exceptional situations (e.g., error conditions, usually modeled as extended use cases [5]) or objects that are instances of library or framework classes. At the very least, we may want to distinguish the different types of coupling taking place in these different categories.

The choices we make regarding the entity, granularity, and scope of measurement depend on how we intend to apply dynamic coupling. Such choices form a classification of dynamic coupling measures that is summarized in Table 1.

2.2 Definitions

Before defining dynamic coupling measures, we introduce below the formal framework that will allow us to provide precise and unambiguous definitions. Not only do such definitions ensure that the reader understands the measures precisely, but they are also easily amenable to the analysis of their properties and facilitate the development of a dynamic analyzer by providing precise specifications. We provide a set of generic definitions that are based on the data model in Fig. 2, which models the type of information to be collected. Each class and association in the class diagram corresponds to a set and a mathematical relation, respectively. The inheritance relationship corresponds to a set partition. Based on this, we define the measures using set theory and first order logic.

A few details of the class diagram in Fig. 2 need to be discussed. Most role names are not shown, to avoid unnecessary cluttering of the class diagram. When no role name is provided, the meaning of associations is quite clear from the source and target classes. For example, methods are defined in a class, method invocations consist of a caller method in a source class and a callee method in a target class. Some of the key attributes are shown. One notable detail is that the line number where the target method is invoked is an attribute of a message that serves to uniquely identify it, as specified by the OCL[1] constraint shown in the class diagram. This is necessary because the same target method may be invoked in different statements and control flow paths in the same source method. Messages bearing those different invocations are considered distinct because they are considered to provide different contexts of invocation for the method.

Furthermore, associations with role names `caller`, `source`, and `sender` should show an `{exclusive or}` constraint dependency to associations with role names `callee`, `target`, and `receiver`, respectively. These constraints are not shown to avoid cluttering the diagram but are important as, in our context, distinct methods, classes, and objects must be involved in the links corresponding to those associations. In other words, in the context of our coupling measurement, method invocations are linked to two distinct class instances and two distinct method instances and messages involve two distinct objects. As expected, method invocations between classes are differentiated from messages between objects. A method name and signature uniquely identifies a method in the context of a specific class and a method invocation must be clearly linked to a method. This is why `MethInvocation` has associations with both `Class` and `Method`.

2.2.1 Sets

The first step is to define the basic sets on which to build our definitions. These sets are derived from the data model in Fig. 2.

- C: Set of classes in the system. C can be partitioned into the subsets of application classes (AC), library classes (LC), and framework classes (FC). Some of these subsets may be empty, $\mathrm{C} = \mathrm{AC} \cup \mathrm{LC} \cup \mathrm{FC}$ and

1. The Object Constraint Language (OCL) [28] is mostly used to specify constraints on class diagrams, operation pre/postconditions, and class invariants.

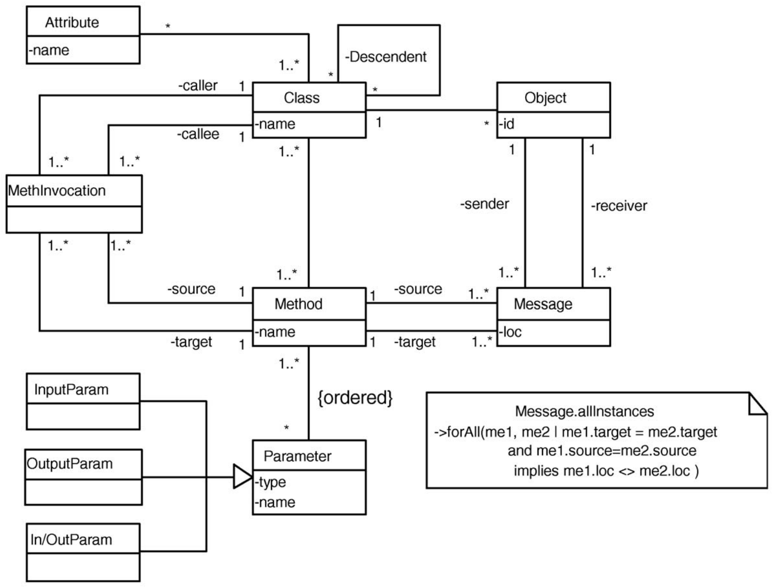

Fig. 2. Class diagram capturing a data model of the dynamic analysis information.

$AC \cap LC \cap FC = \emptyset$. Distinguishing such subsets may be important for defining the scope of measurement, as discussed above.

- O: Set of objects instantiated by the system while executing all scenarios of all use cases (including exceptional use cases, e.g., treating error conditions, which are usually modeled as use cases extending base use cases).
- M: Set of methods in the system (as identified by their signature).
- Lines of code are defined on the set of natural numbers (N).

2.2.2 Relations

We now introduce mathematical relations on the sets that are fundamental to the definitions of our measures.

- D and A are relations *onto* $C(\subseteq C \times C)$. D is the set of descendent classes of a class and A is the set of ancestors of a class.
- ME is the set of possible messages in the system: $ME \subseteq O \times M \times N \times O \times M$. Indicated by the domain of ME, a message is described by a source object and method sending the message, a line of code (N), and a target object and method. Note that the sending of a message may not only correspond to a method invocation, but also to the sending of a signal [5]. The message is then asynchronous and on receipt of the signal, the target object triggers the execution of the target method. In Java, an active object (with its own thread of control) would typically have a `run()` method reading from a queue of signal objects and invoke the appropriate method after reading the next signal in the queue.
- IV is the set of possible method invocations in the system: $IV \subseteq M \times C \times M \times C$. An invocation is characterized by the invoking class and method and the class and method being invoked.
- Other binary relations will be used in the text and their semantics can be easily derived from their domain and are denoted R_{Domain}. For example, $R_{MC} \subseteq M \times C$ refers to methods being defined in classes, a binary relation from the set of methods to the set of classes.

2.2.3 Consistency Rule

The relations IV and ME play a fundamental role in all our measures. In practice, an analysis of sequence diagrams or a dynamic analysis of the code allows us to construct ME. From that information, IV must be derived, but this is not trivial as polymorphism and dynamic binding tend to complicate the mapping. The consistency rule below specifies the dependencies between the two relations and can be used to develop algorithms that derive IV from ME.

$$\begin{aligned}
&(\exists(o_1, c_1), (o_2, c_2) \in R_{OC})(\exists l \in N)(o_1, m_1, l, o_2, m_2) \in ME \Rightarrow \\
&(\exists c_3 \in A(c_1) \cup \{c_1\}, c_4 \in A(c_2) \cup \{c_2\}) \\
&((m_1, c_3) \in R_{MC} \wedge ((\forall c_5 \in A(c_1) - \{c3\})(m_1, c_5) \\
&\quad \in R_{MC} \Rightarrow c_5 \in A(c_3))) \wedge \\
&((m_2, c_4) \in R_{MC}) \wedge ((\forall c_6 \in A(c_2) - \{c_4\})(m_2, c_6) \\
&\quad \in R_{MC} \Rightarrow c_6 \in A(c_4))) \wedge \\
&(m_1, c_3, m_2, c_4) \in IV.
\end{aligned}$$

2.2.4 Working Example

We now use a small working example, as shown in Fig. 3, to illustrate the definitions above. Though it is assumed that our measures are collected through static and dynamic analysis of code, we use UML to describe a fictitious example, because it is more legible than pseudocode. This example is designed to illustrate the subtleties arising from polymorphism and dynamic binding. Other aspects, such as method signatures, have been intentionally kept simple to focus on polymorphism and dynamic binding.

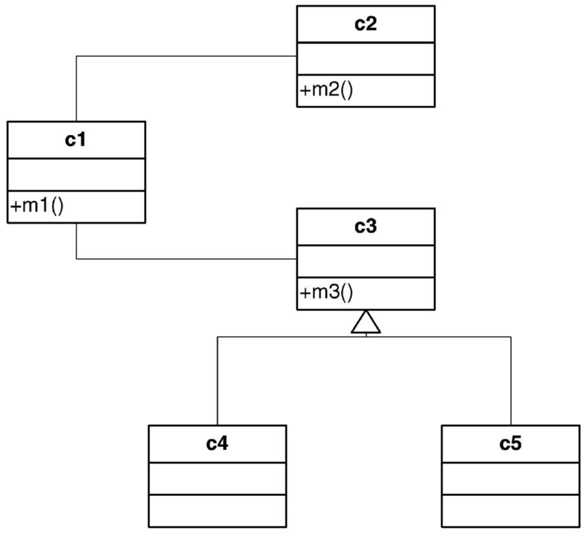

Fig. 3. Working class diagram example (UML notation).

The following sets can be derived from Fig. 3:

$$C = \{c1, c2, c3, c4, c5\}$$
$$M = \{m1, m2, m3\}$$
$$R_{MC} = \{(m1, c1), (m2, c2), (m3, c3)\}.$$

In order to derive other relevant sets and relations, let us introduce the sequence diagrams in Fig. 4, where each message is numbered. As our fictitious example is represented with UML diagrams, objects are referred to by using the sequence diagram number where they appear and their own identification number (i.e., $\mathtt{SD_i{:}\,objectid}$). Similarly, we denote the line of code of the method invocation in message tuples as $\mathtt{l(SD_i{:}\,messageid)}$. In the example, we assume that the line of code of the method invocations `m3()` in messages $\mathtt{SD_1{:}\,1.1}$, $\mathtt{SD_1{:}\,1.2}$, and $\mathtt{SD_1{:}\,1.3}$ are different. Furthermore, since the sequence diagrams do not specify the sender object, source class and source method of the method invocations `m1()` in messages $\mathtt{SD_1{:}\,1}$ and $\mathtt{SD_2{:}\,1}$, the example sets derived below account for only the four (completely specified) messages $\mathtt{SD_1{:}\,1.1}$, $\mathtt{SD_1{:}\,1.2}$, $\mathtt{SD_1{:}\,1.3}$, and $\mathtt{SD_2{:}\,1.1}$:

$$O = \{SD_1 : 1, SD_1 : 2, SD_1 : 3, SD_2 : 1, SD_2 : 2\}$$
$$R_{OC} = \{(SD_1 : 1, c1), (SD_1 : 2, c4), (SD_1 : 3, c5), (SD_2 : 1, c1), (SD_2 : 2, c2)\}$$
$$ME = \{(SD_1 : 1, m1, l(SD_1 : 1.1), SD_1 : 2, m3), (SD_1 : 1, m1, l(SD_1 : 1.2), SD_1 : 3, m3), (SD_1 : 1, m1, l(SD_1 : 1.3), SD_1 : 3, m3), (SD_2 : 1, m1, l(SD_2 : 1.1), SD_2 : 2, m2)$$
$$IV = \{(m1, c1, m3, c3), (m1, c1, m2, c2)\}.$$

2.2.5 Definitions of Measures

The measures are all defined as cardinalities of specific sets. They are therefore defined on an absolute scale and are amenable, as far as measurement theory is concerned, to the type of regression analysis performed in Section 4. Those sets are defined below and are given self-explanatory names, following the notation summarized in Table 2. First, as mentioned above, we differentiate the cases where the entity of measurement is the object or the class. Second, as in previous *static* coupling frameworks [10], we differentiate *import* from *export* coupling, that is the *direction* of coupling for a class or object. For example, we differentiate whether a method executed on an object calls (imports) or is called by (exports) another object's method. Furthermore, orthogonal to the entity of measurement and direction of coupling considered, there are at least three different ways in which the *strength* of coupling can be measured. First, we provide definitions for import and export coupling when the entity of measurement is the object and the granularity level is the class. Phrases outside and between parentheses capture the situations for import and export coupling, respectively.

- *Dynamic messages*. Within a runtime session, it is possible to count the total number of *distinct messages* sent from (received by) one object to (from) other objects, within the scope considered. That information is then aggregated for all the objects of each class. Two messages are considered to be the same if

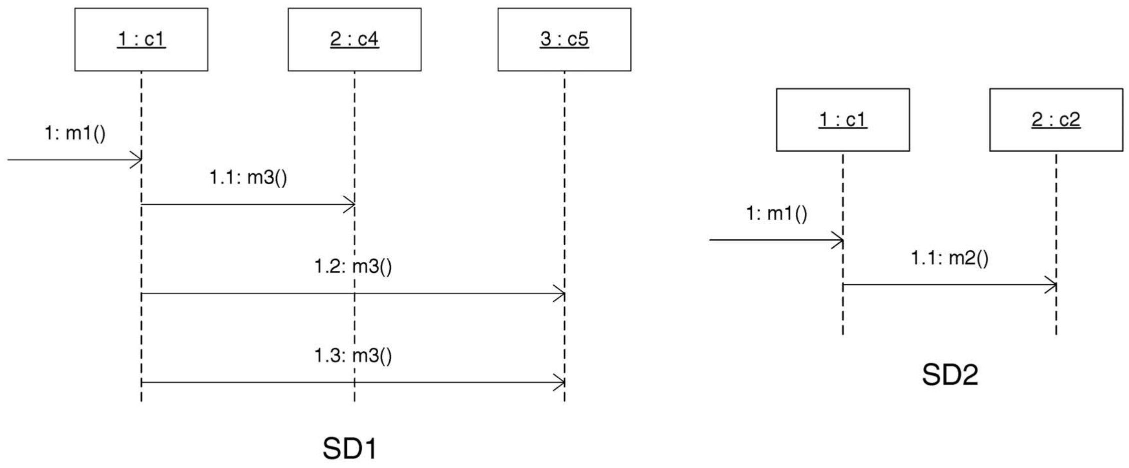

Fig. 4. Two hypothetical sequence diagrams related to Fig. 3.

TABLE 2
Summary of Dynamic Coupling Measures

Direction	Entity of Measurement	Strength	Set Definition
Import **C**oupling	**O**bject	**D**ynamic messages	$IC\_OD(c_1) = \{(m_1, c_1, l, m_2, c_2) \mid (\forall(o_1, c_1) \in R_{OC})\ (\exists\ (o_2, c_2) \in R_{OC}, l \in N)$ $c_1 \neq c_2 \wedge (o_1, m_1, l, o_2, m_2) \in ME\}$
		Distinct **M**ethods	$IC\_OM(c_1) = \{(m_1, c_1, m_2, c_2) \mid (\forall(o_1, c_1) \in R_{OC})\ (\exists\ (o_2, c_2) \in R_{OC}, l \in N)$ $c_1 \neq c2 \wedge (o_1, m_1, l, o2, m2) \in ME\}$
		Distinct **C**lasses	$IC\_OC(c_1) = \{(m_1, c_1, c_2) \mid (\forall(o_1, c_1) \in R_{OC})\ (\exists\ (o_2, c_2) \in R_{OC}, l \in N)$ $c_1 \neq c_2 \wedge (o_1, m_1, l, o_2, m_2) \in ME\}$
	Class	**D**ynamic messages	$IC\_CD(c_1) = \{(m_1, c_1, l, m_2, c_2) \mid (\exists\ (o_3, c_3), (o_4, c_4) \in R_{OC})\ (\exists\ l \in N)$ $c_1 \neq c_2 \wedge\ (o_3, m_1, l, o_4, m_2) \in ME \wedge$ $(\exists\ c_1 \in A(c_3) \cup \{c_3\}, c_2 \in A(c_4) \cup \{c_4\})$ $((m_1, c_1) \in R_{MC} \wedge ((\forall c_5 \in A(c_1) - \{c_1\})\ (m_1, c_5) \in R_{MC} \Rightarrow$ $c_5 \in A(c_1))) \wedge ((m_2, c_2) \in R_{MC}) \wedge ((\forall c_6 \in A(c_4) - \{c_2\})$ $(m_2, c_6) \in R_{MC} \Rightarrow c_6 \in A(c_2))) \wedge (m_1, c_1, m_2, c_2) \in IV\}$
		Distinct **M**ethods	$IC\_CM(c_1) = \{(m_1, c_1, m_2, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC})$ $c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\}$
		Distinct **C**lasses	$IC\_CC(c_1) = \{(m_1, c_1, c_2) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC})$ $c_1 \neq c_2 \wedge (m_1, c_1, m_2, c_2) \in IV\}$
Export **C**oupling	**O**bject	**D**ynamic messages	$EC\_OD(c_1) = \{(m_2, c_2, l, m_1, c_1) \mid (\forall(o_1, c_1) \in R_{OC})\ (\exists\ (o_2, c_2) \in R_{OC}, l \in N)$ $c_1 \neq c_2 \wedge (o_2, m_2, l, o_1, m_1) \in ME\}$
		Distinct **M**ethods	$EC\_OM(c_1) = \{(m_2, c_2, m_1, c_1) \mid (\forall(o_1, c_1) \in R_{OC})\ (\exists\ (o_2, c_2) \in R_{OC}, l \in N)$ $c_1 \neq c_2 \wedge (o_2, m_2, l, o_1, m_1) \in ME\}$
		Distinct **C**lasses	$EC\_OC(c_1) = \{(m_2, c_2, c_1)\ (\forall(o_1, c_1) \in R_{OC})\ (\exists\ (o_2, c_2) \in R_{OC}, l \in N)$ $c_1 \neq c_2 \wedge (o_2, m_2, l, o_1, m_1) \in ME\}$
	Class	**D**ynamic messages	$EC\_CD(c_1) = \{(m_2, c_2, l, m, c) \mid (\exists\ (o_3, c_3), (o_4, c_4) \in R_{OC})\ (\exists\ l \in N)$ $c_1 \neq c_2 \wedge (o_4, m_2, l, o_3, m_1) \in ME \wedge$ $(\exists\ c_1 \in A(c_3) \cup \{c_3\}, c_2 \in A(c_4) \cup \{c_4\})$ $((m_1, c_1) \in R_{MC} \wedge ((\forall c_5 \in A(c_3) - \{c\})\ (m_1, c_5) \in R_{MC} \Rightarrow$ $c_5 \in A(c_1))) \wedge$ $((m_2, c_2) \in R_{MC}) \wedge ((\forall c_6 \in A(c_4) - \{c_2\})\ (m_2, c_6) \in R_{MC} \Rightarrow$ $c_6 \in A(c_2))) \wedge$ $(m_2, c_2, m_1, c_1) \in IV\}$
		Distinct **M**ethods	$EC\_CM(c_1) = \{(m_2, c_2, m_1, c_1) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC})$ $c_1 \neq c_2 \wedge (m_2, c_2, m_1, c_1) \in IV\}$
		Distinct **C**lasses	$EC\_CC(c_1) = \{(m_2, c_2, c_1) \mid (\exists\ (m_1, c_1), (m_2, c_2) \in R_{MC})$ $c_1 \neq c_2 \wedge (m_2, c_2, m_1, c_1) \in IV\}$

their source and target classes, the method invoked in the target class, and the statement from which it is invoked in the source class are the same. The latter condition reflects the fact that a different context of invocation is considered to imply a different message. In a UML sequence diagram, this would be represented as distinct messages with identical method invocations but different guard conditions.

- *Distinct method invocations*. A simpler alternative is to count the number of *distinct* methods invoked by each method in each object (that invoke methods in each object). Note that this is different from simply counting method invocations as we count each distinct method only once. That information is then aggregated for all the objects of each class.
- *Distinct classes*. It is also possible to count only the number of distinct server (client) classes that a method in a given object uses (is used by). That information is then aggregated for all the objects of each class.

If we now look at where the calling and called methods are defined and implemented, the entity of measurement is the class and we can provide similar definitions. We then count the number of distinct messages originating from (triggering the executions of) methods in the class, the number of distinct methods invoked by (that invoke) the class methods, and the number of distinct classes from which the class is using methods (that uses its methods).

Table 2 shows the formal set definitions of the measures when the granularity is the class, and the scope is the system. We provide an intuitive textual explanation only for the first set: $IC\_OM(c)$. Other sets can be interpreted in a similar manner.

$IC\_OM(c)$: A set containing all tuples *(source method, source class, target method, target class)* such that there exists an object *o* instantiating *c* (whose coupling is being measured) that sends a message to at least one instance of the *target class* in order to trigger the execution of the *target method*. The corresponding metric is simply the cardinality of this set. Note that the source class must be different from the target class ($c_1 \neq c_2$), because we are focusing on dependencies that contribute to coupling between classes, not their cohesion (as further discussed in [9], [10]). Reflexive method invocations are therefore excluded.

TABLE 3
Example Coupling Sets When the Entity of Measurement is the Class

IC_CD(c1)	{(m1,c1,I(SD$_1$:1.1),m3,c3),(m1,c1,I(SD$_1$:1.2),m3,c3),(m1,c1,I(SD$_1$:1.3),m3,c3),(m1,c1,I(SD$_2$:1.1),m2,c2)}
IC_CM(c1)	{(m1,c1,m3,c3), (m1,c1,m2,c2)}
IC_CC(c1)	{(m1,c1,c3), (m1,c1,c2)}
EC_CD(c2)	{(m1,c1,I(SD$_2$:1.1),m2,c2)}
EC_CM(c2)	{(m1,c1,m2,c2)}
EC_CC(c2)	{(m1,c1,c2)}
EC_CD(c3)	{(m1,c1,I(SD$_1$:1.1),m3,c3), (m1,c1,I(SD$_1$:1.2),m3,c3), (m1,c1,I(SD$_1$:1.3),m3,c3)}
EC_CM(c3)	{(m1,c1,m3,c3)}
EC_CC(c3)	{(m1,c1,c3)}

TABLE 4
Example Coupling Sets When the Entity of Measurement is the Object

IC_OD(c1)	{(m1,c1,I(SD$_1$:1.1),m3,c4),(m1,c1,I(SD$_1$:1.2),m3,c5),(m1,c1,I(SD$_1$:1.3),m3,c5),(m1,c1,I(SD$_2$:1.1),m2,c2)}
IC_OM(c1)	{(m1,c1,m3,c4), (m1,c1,m3,c5), (m1,c1,m2,c2)}
IC_OC(c1)	{(m1,c1,c4), (m1,c1,c5), (m1,c1,c2)}
EC_OD(c2)	{(m1,c1,I(SD$_2$:1.1),m2,c2)}
EC_OM(c2)	{(m1,c1,m2,c2)}
EC_OC(c2)	{(m1,c1,c2)}
EC_OD(c4)	{(m1,c1,I(SD$_1$:1.1),m3,c4)}
EC_OM(c4)	{(m1,c1,m3,c4)}
EC_OC(c4)	{(m1,c1,c4)}
EC_OD(c5)	{(m1,c1,I(SD$_1$:1.2),m3,c5), (m1,c1,I(SD$_1$:1.3),m3,c5)}
EC_OM(c5)	{(m1,c1,m3,c5)}
EC_OC(c5)	{(m1,c1,c5)}

TABLE 5
Changed Import Coupling Sets after Adding a New Implementation of `m3()` in `c5`

IC_CD(c1)	{(m1,c1,I(SD$_1$:1.1),m3,c3), ~~(m1,c1,I(SD$_1$:1.2),m3,c3),(m1,c1,I(SD$_1$:1.3),m3,c3),~~ **(m1,c1,I(SD$_1$:1.2),m3,c5)**, **(m1,c1,I(SD$_1$:1.3),m3,c5)**, (m1,c1, I(SD$_2$:1.1),m2,c2)}
IC_CM(c1)	{(m1,c1,m3,c3), **(m1,c1,m3,c5)**, (m1,c1,m2,c2)}
IC_CC(c1)	{(m1,c1,c3), **(m1,c1,c5)**, (m1,c1,c2)}

2.2.6 Higher Granularities

If we want to measure dynamic coupling at higher levels of granularity, this can be easily defined by performing the union of the coupling sets of a set of classes or objects, depending on the entity of measurement. For example, if the entity of measurement is the class and the level of granularity is the subsystem, then for each subsystem *SS* there corresponds a subset of classes that it contains, $\mathrm{SC} \in 2^{\mathrm{C}}$, and we can define:

$$\mathrm{IC\_CM(SS)} = \cup_{(\mathrm{all\ c} \in \mathrm{SC})}\mathrm{IC\_CM(c)}.$$

Similarly, when the entity of measurement is the object: For each use case *UC* there corresponds a set of participating objects $\mathrm{SO} \in 2^{\mathrm{O}}$ (that are involved in the UC's sequence diagram(s)), and we can define:

$$\mathrm{IC\_CM(UC)} = \cup_{(\mathrm{all\ o} \in \mathrm{SO})}\mathrm{IC\_CM(o)}.$$

Similar definitions can be provided for all levels of granularity.

2.2.7 Example

Returning to our working example in Figs. 3 and 4, we provide below all the nonempty coupling sets. When the entity of measurement as well as the granularity is the class, we obtain the import and export coupling sets illustrated in Table 3. When the entity of measurement is the object, and the granularity is the class, we obtain the coupling sets in Table 4. The export coupling sets for `c1` as well as the import coupling sets for `c2`, `c3`, `c4`, and `c5` are empty.

To gain a better insight into the impact of polymorphism on coupling, let us change the class diagram in Fig. 3 by adding a new implementation of method `m3()` in `c5`: $\mathrm{R_{MC}} = \{(\mathrm{m1, c1}), (\mathrm{m3, c3}), (\mathbf{m3, c5}), (\mathrm{m2, c2})\}$, while keeping the sequence diagrams in Fig. 4 unchanged. This results in a new element in IV:

$$\mathrm{IV} = \{(\mathrm{m1, c1, m3, c3}), (\mathbf{m1, c1, m3, c5}), (\mathrm{m1, c1, m2, c2})\}.$$

The other sets (C, M, O, $\mathrm{R_{OC}}$, and ME) remain unchanged. When the entity of measurement is the class, the new method implementation results in significantly changed import coupling sets for class `c1` (see Table 5, where removed elements are struck through, whereas new elements are bolded). Adding a new implementation of an existing method in a subclass has resulted in increased import coupling for class `c1`. This is because class `c1` now imports from one additional class (`c5`), one additional method (`m3()` in `c5`), and one additional distinct method invocation. However, object import coupling $(\mathrm{IC\_Ox(c)})$

TABLE 6
Changed Export Coupling Sets after Adding a New Implementation of `m3()` in `c5`

EC_CD(c2)	{(m1,c1,l(SD_2:1.1), m2,c2)}
EC_CM(c2)	{(m1,c1,m2,c2)}
EC_CC(c2)	{(m1,c1,c2)}
EC_CD(c3)	{(m1,c1,l(SD_1:1.1), m3,c3), ~~(m1,c1,l(SD_1:1.2), m3,c3)~~, ~~(m1,c1,l(SD_1:1.3),m3,c3)~~}
EC_CM(c3)	{(m1,c1,m3,c3)}
EC_CC(c3)	{(m1,c1,c3)}
EC_CD(c5)	{**(m1,c1,l(SD_1:1.2),m3,c5)**, **(m1,c1,l(SD_1:1.3),m3,c5)**}
EC_CM(c5)	{**(m1,c1,m3,c5)**}
EC_CC(c5)	{**(m1,c1,c5)**}

remains unchanged, as at the object level, instances of `c1` were already importing from `c5`.

In a similar way, the export coupling of class `c3` has decreased and the export coupling of class `c5` has increased (see Table 6).

2.3 Analysis of Properties

We show here that the five coupling properties presented in [10] are valid for our dynamic coupling measures. The motivation is to perform an initial theoretical validation by demonstrating that our measures have intuitive properties that can be justified. We use IC_OM and IC_CM at the lowest granularity level (object, class) and system level as examples, but the demonstrations[2] below can be performed in a similar way for all coupling measures, at all levels of granularity.

Nonnegativity. It is not possible for the dynamic coupling measures to be negative because they measure the cardinality of sets, e.g., IC_OM returns a set of tuples $(\mathrm{m, c, m', c'}) \in \mathrm{M \times C \times M \times C}$.

Null values. At the system level, if S is the set that includes all the objects that participate in all the use cases of the system, $IC\_OM(S)$ is empty (and coupling equal to 0) if and only if the set of messages in S is empty:

$$\mathrm{ME} = \emptyset \Leftrightarrow \mathrm{IC\_OM(S)} = \emptyset.$$

This is consistent with our intuition as this should be the only case where we get a null coupling value. Since $\mathrm{ME} = \emptyset \Leftrightarrow \mathrm{IV} = \emptyset$ (consistency rule), we also have:

$$\mathrm{ME} = \emptyset \Leftrightarrow \mathrm{IC\_CM(S)} = \emptyset.$$

At the object level, for $IC\_OM(o)$, we have:

$$(\forall\, \mathrm{o} \in \mathrm{O}, \mathrm{m} \in \mathrm{M}, \mathrm{l} \in \mathrm{N}, \mathrm{o'} \in \mathrm{O}, \mathrm{m'} \in \mathrm{M})$$
$$(\mathrm{o, m, l, o', m'}) \notin \mathrm{ME} \Leftrightarrow \mathrm{IC\_OM(o)} = \emptyset.$$

Again, this is intuitive, as we should only obtain a null value if and only if object o does not participate in any message as sender or receiver. Similarly, at the class level, we obtain:

$$(\forall \mathrm{o} \in \mathrm{O}, \mathrm{c} \in \mathrm{C}, (\mathrm{o, c}) \in \mathrm{R_{oc}})\mathrm{IC\_OM(o)} = \emptyset$$
$$\Leftrightarrow IC\_CM(c) = \emptyset \text{ (consistency rule)}.$$

Monotonicity. If a class c is modified such that at least one instance o sends/receives more messages, its import/export coupling can only increase or stay the same, for any of the coupling measures defined above.

If object $\mathrm{o} \in \mathrm{O}$ sends an additional message $(\mathrm{o, m, l, o', m'})$ $\in \mathrm{ME}$, this cannot reduce the number of pairs $(\mathrm{method, class})$ $\in \mathrm{R_{MC}}$ that are part of the sets $IC\_OM(o)$ or $IC\_OM(S)$. The same can be said for export coupling if object $\mathrm{o} \in \mathrm{O}$ receives an additional message.

Adding a message to ME may or may not lead to a new method invocation in IV. But, even if this is the case, the sets $IC\_CM(c)$ and $IC\_CM(S)$ cannot possibly lose any elements.

Similar arguments can be provided for all coupling measures, at all levels of granularity. To conclude, by adding messages and method invocations in a system, object and class coupling measures cannot decrease, respectively, thus complying with the monotonicity property.

Impact of merging classes. Assuming c' is the result of merging c_1 and c_2, thus transforming system S into S', for any *Coupling* measure, we want the following properties to hold at the class and system levels:

$$\mathrm{Coupling(c_1) + Coupling(c_2) \geq Coupling(c')}$$
$$\mathrm{Coupling(S) \geq Coupling(S')}.$$

Taking IC_CD as an example, we can easily show this property holds: All instances of c_1 and c_2 in IV's tuples are substituted with c'. If there exist tuples of the type $(\mathrm{m_1, c_1, m_2, c_2})$ in IV, then they are transformed into tuples of the form $(\mathrm{m_1, c', m_2, c'})$. For IC_Cx measures, since we exclude reflexive method invocations because they do not contribute to coupling (Section 2.2), then tuples of the form $(\mathrm{m_1, c', m_2, c'})$ disappear because of the merging. Hence:

$$|\mathrm{IC\_CD(c')}| \leq |\mathrm{IC\_CD(c_1)}| + |\mathrm{IC\_CD(c_2)}|.$$

Similar arguments can be made for all other coupling measures.

Merging uncoupled classes. Following reasoning similar to that above, if two classes c_1 and c_2 do not have any coupling, this means there is no tuple of the type $(\mathrm{m_1, c_1, m_2, c_2})$ in IV. If we merge them into one class, we therefore cannot obtain tuples of the type $(\mathrm{m_1, c', m_2, c'})$.

2. These demonstrations are admittedly rather informal. We adopted a level of formality that we deemed sufficient to convince the reader these properties did indeed hold, without making the discussion unnecessarily terse.

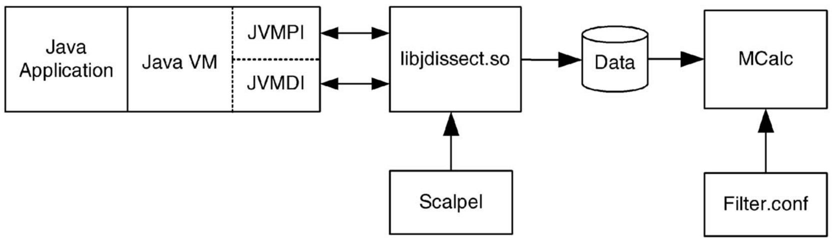

Fig. 5. Architecture of the JDissect tool.

Then, we can conclude IC_CD fulfills the following property:

$$|\mathrm{IC\_CD}(c')| = |\mathrm{IC\_CD}(c_1)| + |\mathrm{IC\_CD}(c_2)|.$$

This property also holds for all other coupling measures.

Symmetry between export and import coupling. By symmetry, for all class level dynamic coupling measures, we infer that the following property holds:

$$\cup_{(\text{all } c \in C)} \mathrm{EC\_Cx}(c) = \cup_{(\text{all } c \in C)} \mathrm{IC\_Cx}(c).$$

This stems from the fact that for any $(m, c, m', c') \in IV$, there is always a $l \in N$ such that $(m, c, l, m', c') \in \mathrm{EC\_CD}(c')$ and $(m, c, l, m', c') \in \mathrm{IC\_CD}(c)$. Along the same lines, for each $(m, c, m', c') \in \mathrm{IC\_CM}(c)$ and $(m, c, c') \in \mathrm{IC\_CC}(c)$, there is a corresponding $(m, c, m', c') \in \mathrm{EC\_CM}(c')$ and $(m, c, c') \in \mathrm{EC\_CC}(c')$, respectively.

Following a similar argument when the entity of measurement is the object, we obtain:

$$\cup_{(\text{all } o \in O)} \mathrm{EC\_Ox}(o) = \cup_{(\text{all } o \in O)} \mathrm{IC\_Ox}(o).$$

The symmetry property is intuitive because anything imported by a class or object has to be exported by another class or object, respectively. This condition applies at all levels of granularity.

Based on the property analysis above, we can see that our coupling measures seem to exhibit intuitive properties that would be expected when measuring coupling. This constitutes a theoretical validation of the measures. Section 4 focuses on their empirical validation, using project data.

3 Collecting Dynamic Coupling Data

It is crucial to collect dynamic coupling data in a practical and efficient manner. This section describes two alternative approaches. The first is based on collecting the coupling data from executing programs, whereas the second calculates the measures based on dynamic UML models.

3.1 Tool for Collecting Dynamic Coupling Measures at Runtime

To collect dynamic coupling data from Java applications, we developed a tool: JDissect. An overview of the architecture is depicted in Fig. 5. The tool separates the collection and analysis of dynamic coupling data into two phases. In the first phase, data from a running Java program is gathered and stored. This is accomplished by having the Java Virtual Machine (JVM) load a library of data collection routines (`libjdissect.so`) that are called whenever specified internal events occur. The interfaces used for communication between the JVM and the library are called JVMPI (Java VM Profiling Interface) and JVMDI (Java VM Debugging Interface). Most of the data is collected from the profiling interface. The JVMDI is used to obtain the unique line number from which a method call originates (to obtain the information needed to calculate the xx_xD measures). During the data collection phase, a user may interactively tag messages belonging to specific scenarios or use cases through a separate utility (`Scalpel`) that communicates with `libjdissect.so` through a socket connection. These tags can subsequently be used to limit the scope of measurement (e.g., to specific use cases) and, potentially, to compute measures at higher levels of granularity than the class (e.g., at the use case aggregation level). During the data collection process, the library populates a data structure as specified in Fig. 2. When the application terminates, the data is stored in a flat file structure (`Data`).

In the second phase, the data is analyzed. Another executable (`MCalc`), sharing a great deal of code with the library, reads the flat files into a data structure identical to that used by the library. This structure is analyzed to obtain the dynamic coupling measures. The analysis tool traverses the data structure in Fig. 2 and computes the sets specified in Table 2. A configuration file (`Filter.conf`) can be used to limit the scope of measurement, e.g., excluding library or framework classes. Each measure is then computed simply by counting the number of elements in each set. Data from several runtime sessions can be merged by the analysis tool, such that accumulated dynamic coupling data can be computed. This merging capability enables the collection of coupling data for Java systems for which several concurrent instances of the JVM are used, such as large, distributed, or component-based systems.

Our coupling tool utilizes interfaces provided by the Java Virtual Machine to collect the message traces and other information specified in Fig. 2. Another possible approach could have been to *instrument* the system. Instrumentation can be done either at the *source code* or *byte code* level using tools such as the Java Compiler Compiler (JavaCC) [22] or the Byte Code Engineering Library (BCEL) [21], respectively. However, utilizing the existing interfaces to the Java VM provides several benefits over instrumentation. Instrumenting the code means that we are testing the instrumented version and not the actual version, which may lead to different outputs and system states. Since instrumentation causes a significant effort overhead, if the system is evolving rapidly, the project manager will also be reluctant to keep instrumenting the new versions.

Furthermore, source code instrumentation requires access to the Java application source code. This might be a disadvantage in cases where an application uses libraries for which the source code is not available. Finally, instrumentation might cause a significant performance overhead. In contrast to our approach, both source code and byte code instrumentation require that parts of the data collection software be written in Java. Subsequently, the byte code of the data collection software is interpreted by the Java VM. Since our data collection tool is written in C++ and dynamically linked with the JVM at runtime, there is probably less performance overhead associated with our approach than with data collection tools employing instrumentation. As performance overhead increases, the behavior of concurrent software is more likely to be affected by the data collection process and it is important to minimize the chances of such a problem occurring.

3.2 Using UML Models for Data Collection

So far, we have assumed that dynamic coupling data are collected through dynamic analysis of the code. It was also suggested that it might be possible to collect the dynamic coupling data through analysis of dynamic UML models, e.g., interaction diagrams. Measuring coupling on early design artifacts would be of practical importance because one could use that information for early decision making. For example, assuming that the necessary UML diagrams are available for a given design, one could derive test cases [8] and compute the dynamic coupling associated with each of the test cases (use case scenarios) based on the UML diagrams. For example, test cases with high coupling could be exercised first, as they would be expected to uncover more faults and, therefore, the test plan would provide an order in which to run test cases based on dynamic coupling information.

When measuring dynamic coupling based on UML models, the main problem lies with interaction diagrams. If we resort to UML diagrams for dynamic coupling measurement, we have to find a substitute for the line of code where the invocation is located to distinguish messages (in ME) and compute xx_xD measures. A natural substitute is the guard or path condition (which must be true for a message to be sent), which corresponds to different contexts of invocations. An identical method on two messages with two distinct guard conditions must correspond to different invocation statements in the code. However, one guard condition on a message does not have to correspond to one invocation statement in the code. For example, one may have a guard of the form `[A or B]` that triggers the invocation of `m()`, and the corresponding code may show two distinct invocations statements for `m()`, each of them being in the body of an if statement with conditions `A` and `B`, respectively.

What this implies is that if xx_xD measures are collected from UML interaction diagrams, coupling will tend to be underestimated, because distinct elements of ME will not be distinguishable using UML interaction diagrams. However, the question is whether, in practice, this makes any significant difference. The advantages of using dynamic coupling measures on early UML artifacts may outweigh the drawbacks that are due to their lower precision. Furthermore, xx_xC and xx_xM measures are not affected by the use of UML interaction diagrams. If empirical investigation finds these latter measures to be strongly correlated with xx_xD, it is doubtful the data collection inaccuracy discussed above will have any practical effect.

4 Case Study

This section presents the results of a case study whose objectives are to provide a first empirical validation of the dynamic coupling measures presented above. The first subsection explains in more detail our objectives, the study settings, and the methodology we follow. In subsequent sections quantitative results are presented and interpreted.

4.1 Objectives and Methodology

We selected an open-source software system called Velocity to evaluate the dynamic coupling measures. Velocity is part of the Apache Jakarta Project [21]. Velocity can be used to generate web pages, SQL, PostScript, and other outputs from template documents. It can be used either as a standalone utility or as an integrated component of other systems. A total of 17 consecutive versions (versions 1.0b1 to version 1.3.1) of Velocity were available for analysis. The versions were released within a time span of approximately two years. The versions used in the actual analysis were four subsequent subreleases (called "release candidates" in Velocity) within one major release of the Velocity system (version 1.2). The first subrelease, 1.2rc1, consists of 17,210 source lines of code (SLOC) in 136 core application classes (after removing "dead" code and classes related to test cases, as described further in Section 4.2) in addition to 408 library classes. There were 65 inheritance relationships and 149 instances of method overriding in the first release candidate, thus showing substantial use of polymorphism and dynamic binding. Further descriptive statistics of the classes are provided in [4].

Several types of data were collected from the system. First, change data (i.e., using a class-level source code *diff*) was collected for each application class. Based on the change data, the amount of change (in SLOC added and deleted) of each class within a given set of consecutive versions was computed. Second, to collect the dynamic coupling measures, test cases provided with the Velocity source code were used to exercise each version of the system. Each test case was executed while the JDissect dynamic coupling tracer tool (Section 3.1) computed the dynamic coupling measures. Third, size and a comprehensive set of static coupling measures (a complete list is provided in [4]) were collected using a static code analysis tool. The scope of measurement was the application classes (AC) of Velocity. Thus, coupling to/from library and framework classes were not included (for further details, see Section 2.1).

A first objective of the case study was to determine whether the dynamic coupling measures capture additional dimensions of coupling when compared with static coupling measures. A subsequent, more ambitious objective was to investigate whether dynamic coupling measures are significant indicators of a useful, external quality attribute and are complementary to existing static measures in explaining its variance.

Following the methodology described in [7], we first analyzed the descriptive statistics of the dynamic coupling measures. The motivation was to determine whether they show enough variance and whether some of the properties we expected were visible in the data. The next step was to perform a principal component analysis (PCA), the goal of which was to identify what structural dimensions are captured by the dynamic coupling measures and whether these dimensions are at least partly distinct from static coupling measures. It is usual for software product measures to show strong correlations and for apparently different measures to capture similar structural properties. PCA also helps to interpret what measures actually capture and determine whether all measures are necessary for the purpose at hand. In our case, recall that we want to determine whether all xx_xC, xx_xM, and xx_xD measures are necessary, that is, to what extent they are redundant. Due to size constraints, results from the above analyses are only summarized in this paper and fully reported in [4].

In order to investigate their usefulness as quality indicators, we investigate whether dynamic coupling measures are statistically related to *change proneness*, that is, the extent of change across the versions of the system we used as a case study. To do so, we analyzed the changes (lines of code added and deleted) across the four subreleases of Velocity 1.2. Our goal was to ensure we would only consider *correction* changes as requirements changes are not driven by design characteristics but mainly by external factors. Subreleases in a major release include only correction changes[3] and we were therefore able to factor out requirements changes and obtain more accurate analysis results regarding the impact of coupling on change proneness.

The dependent variable (*Change*) in this study is the total amount of change (source lines of code added and deleted) that has affected each of the 136 application classes participating in the test case executions across the four subreleases of Velocity 1.2. Since none of these classes were added or deleted during the making of the successive releases, the variable *Change* is a measure of the change proneness of these classes. In this case study context, this can be more precisely defined as their tendency to undergo correction changes. Other possible dependent variables could have been selected, such as the number of changes, but we wanted our dependent variable to somehow reflect the extent of changes as well as their frequency.

The above analysis assumes that there is a *cause-effect* relationship between coupling and change proneness, something which is intuitive because classes that strongly depend on or provide services to other classes are more likely to change, through ripple effects, as a result of changes in the system [11]. Predicting the change proneness of a class (i.e., its volatility) can be used to aid design refactoring (e.g., removing "hot-spots"), choosing among design alternatives or assessing changeability decay [1].

3. We checked the change records for the four subreleases of Velocity 1.2 to ensure that this assumption was correct.

One important issue is that not only do we want our measures to relate to change proneness in a statistically significant way, but we want the effect to be additional or complementary to that of *static* coupling measures and class size [7], [19]. If some of the dynamic coupling measures remain statistically significant covariates when the static coupling measures and size measures are included as candidate covariates, this subset of dynamic coupling measures is deemed to significantly contribute to change proneness. We consider this to be empirical evidence of the causal effect between dynamic coupling and change proneness, of their practical usefulness and, hence, we consider it to provide an initial empirical validation of the dynamic coupling measures. More details are provided in Section 4.4.

4.2 Code Coverage

One practical drawback of using dynamic analysis is that one has to ensure that the code is sufficiently exercised to reflect in a complete manner the interactions that can take place between objects. To obtain accurate dynamic coupling data, the complete set of test cases provided with Velocity were used to exercise the system. Though this test suite was supposed to be complete, as it is used for regression test purposes, we used a code coverage tool and discovered that only about 70 percent of the methods were covered by the test cases. A closer inspection of the code revealed that a primary reason for this apparent low coverage was that 34 classes contained "dead" code. In addition, there were many occurrences of alternative constructors and error checking code that were never called. Fortunately, such code does not contribute to coupling. After removing the dead code and filtering out alternative constructors and error checking code, the test cases covered approximately 90 percent of the methods that might contribute to coupling among the application classes in Velocity. Consequently, the code coverage seems to be sufficient to obtain fairly accurate dynamic coupling measures for the 136 "live" application classes of Velocity 1.2.

4.3 Preliminary Analysis Summary

This subsection summarizes the main results from a number of standard, preliminary data analyses that are reported in [4].

4.3.1 *Variability*

We first computed descriptive statistics for coupling and class size measures based on the first sub-release of the studied release (1.2) of Velocity. One notable result is that the mean values for dynamic import coupling measures (e.g., IC_OC) are always equal to the mean values of their corresponding dynamic export coupling measure (e.g., EC_OC). This confirms the symmetry property discussed in Section 2.3. For most measures, there are large differences between the lower 25th percentile, the median, and the 75th percentile, thus showing strong variations in import and export coupling across classes. Many of the measures show a large standard deviation and mean values that are larger than the median values, with a distribution skewed towards larger values. Two of the static coupling measures show (almost) no variation and are not considered in the remainder of the analysis [4]. These measures are related to direct access of public attributes by methods in other classes, which is considered poor practice.

4.3.2 Principal Component Analysis (PCA)

PCA [18] was then used to analyze the covariance structure of the measures and determine the underlying dimensions they capture. Detailed results, provided in [4], show that coupling is divided along four dimensions: IC_Ox, IC_Cx, EC_Ox, and EC_Cx. Thus, all xx_xC, xx_xM, and xx_xD measures belong to identical components when they have identical scope, granularity and entity of measurement, therefore capturing similar properties. This implies that it may be unnecessary to collect all of these measures and, in particular, the xx_xD measures that cannot be collected on UML diagrams and which require expensive dynamic code analysis [4] may not be needed. It is interesting to note that this confirms the PCA results[4] in an earlier case study on a Smalltalk system [2].

Overall, the PCA analysis indicates that our dynamic coupling measures (especially when the entity of measurement is the object) are not redundant with existing static coupling and size measures.

4.3.3 Dynamic Coupling as an Explanatory Variable of Change Proneness

The next step was to analyze the extent to which each of the dynamic coupling measures are related to our dependent variable, change proneness (see Section 4.1). However, since the size (SLOC) of a class is an obvious explanatory variable of *Change* (SLOC added+deleted), it may be more insightful to determine whether a coupling measure is related to change proneness independently of class size. We therefore tested whether the dynamic coupling measures are significant *additional* explanatory variables, over and above what has already been accounted for by size. To achieve this, we systematically performed a multiple linear regression involving class size (SLOC) and each of the dynamic coupling measures and then determined whether the regression coefficient for the coupling measure was statistically significant. Details are reported in [4] and can be summarized as follows: There is strong support for the hypotheses that all dynamic *export* coupling measures are clearly related to change proneness, in addition to what can be explained by size. On the other hand, dynamic *import* coupling measures do not seem to explain additional variation in change proneness, compared to size alone. Once again, this confirms the results obtained in an earlier case study on a Smalltalk system [2]. The following section evaluates the extent to which the dynamic coupling measures are useful predictors when building the best possible models by using size, static coupling, and dynamic coupling measures as possible model covariates.

4.4 Prediction Model of Change Proneness

4.4.1 Model Variables

Throughout this section, the dependent variable is change proneness (see Section 4.1). The independent variables include the size and static coupling measures and our proposed 12 dynamic coupling measures. A complete list of candidate measures is available in [4], and the subset of measures selected in our prediction models are defined in Table 9. Ordinary Least-Squares regression (including outlier analysis) was used to analyze and model the relationship between the independent and dependent variables; that is, between the size/coupling measures of the first subrelease and the amount of changes in the subsequent subreleases. In order to select covariates in our regression model, we use a mixed selection heuristic [20] so as to allow variables to enter, but also to leave, the model when below/above a significance threshold. Though other procedures have been tried (e.g., backward procedure based on variables with highest loadings in principal components), the one we report here yielded models with significantly higher fit.

4.4.2 Rationale for Model Building

Recall that the objective of this regression analysis is to determine whether dynamic coupling measures help to explain additional variation in change proneness, compared to class size (CS) and static coupling alone (see Section 4.1). In other words, we want to determine whether these measures help to obtain a better model fit and, therefore, an improved predictive model. To achieve this objective we proceeded in two steps. First, we analyzed the relationship between *Change* and *CS + Static coupling measures* in order to generate a multivariate regression model that would serve as a baseline of comparison. We then continued by performing multivariate regression, using as candidate covariates all size, static coupling, and dynamic coupling measures. If the goodness of fit of the latter model turns out to be significantly better than the former model we would then be able to conclude that dynamic coupling measures are useful, additional explanatory variables of change proneness.

4.4.3 Discussion of Modeling Results

The first multivariate model we obtained when using size and static coupling measures as candidate covariates is presented in Table 7. After removing one outlier that is clearly overinfluential on the regression results (with an extremely large *Change* value), we obtained a model with three size measures and nine static coupling measures for covariates[5] (for 135 observations). Around 79 percent of the variance in the data set is explained by size and static coupling measures and we obtained an adjusted R^2 of 0.77 (i.e., adjusted for the number of covariates [20]). We do not attempt to discuss the regression coefficients, because such models are inherently difficult to interpret since it is common to see some degree of correlation and interaction between covariates [7]. Smaller, less accurate models (e.g., where covariates are selected based on principal compo-

4. For dynamic coupling measures only, as static measures cannot be collected on Smalltalk systems.

5. Each category of measure is separated by a line in the table, starting with the intercept, static coupling and then class size.

TABLE 7
Regression Model Using Size and Static Coupling as Candidate Covariates

Covariate	Coefficient	Std Error	t Ratio	Prob>|t|
Intercept	14.24	4.08	3.49	0.0007
CBO	2.80	0.78	3.57	0.0005
PIM_EC	1.45	0.22	6.58	<.0001
DAC'	18.87	4.37	4.31	<.0001
OCAEC	-5.36	2.45	-2.18	0.0310
ACMIC	-26.64	6.44	-4.14	<.0001
OCMIC	-12.65	1.01	-12.44	<.0001
OMMIC	4.21	0.50	8.31	<.0001
DMMEC	-2.99	0.58	-5.14	<.0001
OMMEC	-1.41	0.34	-4.12	<.0001
NMD	-1.06	0.28	-3.69	0.0003
NumPara	4.23	0.38	10.87	<.0001
CS2 (semi)	-0.37	0.037	-9.85	<.0001

TABLE 8
Regression Model Using All Measures as Candidate Covariates

Covariate	Coefficient	Std Error	t Ratio	Prob>|t|
Intercept	8.12	3.62	2.24	0.0270
EC_OC	4.32	1.08	4.00	0.0001
EC_OM	-7.70	1.59	-4.81	<.0001
EC_OD	5.02	0.99	5.03	<.0001
IC_CC	-1.14	0.52	-2.19	0.0306
CBO	2.84	0.70	4.02	0.0001
RFC_1	0.67	0.18	3.66	0.0004
RFC	-0.05	0.01	-3.37	0.0010
OCAIC	19.39	4.23	4.58	<.0001
OCMIC	-10.37	0.95	-10.90	<.0001
OMMIC	4.37	0.55	7.90	<.0001
DMMEC	-1.17	0.42	-2.75	0.0069
OMMEC	-1.46	0.25	-5.74	<.0001
AMAIC	6.06	1.98	3.05	0.0028
NMI	4.38	0.98	4.47	<.0001
NMpub	-1.86	0.58	-3.17	0.0019
NumPara	2.60	0.73	3.53	0.0006
CS1 (SLOC)	-0.22	0.02	-9.82	<.0001

nents) would have been easier to interpret but recall that our goal was to demonstrate the usefulness of dynamic coupling measures as predictors of change proneness. Furthermore, analysis results provided in [4] show that, when significant, the relationships are in the expected direction for our dynamic coupling measures.

When including, in the set of candidate covariates, the dynamic coupling measures, we obtain a very different model (Table 8). Four dynamic coupling measures (highlighted in italics), as well as nine static coupling measures and four size measures, were included as covariates in the model (we retained, as for the other model, all covariates with p-values below 0.1). The model explains 87 percent of the variance in the data set and shows an adjusted R^2 of 0.85. Therefore, even when accounting for the difference in number of covariates, the coefficient of determination (R^2) increased by 8 percent or 35 percent of the unexplained variance (from 0.77 to 0.85) when using dynamic coupling measures as candidate covariates. This is an indication that some of the dynamic coupling measures are complementary indicators to static coupling and size measures as far as change proneness is concerned.

It is also interesting to note that three out of the four dynamic coupling measures capture export coupling. One import coupling measure is nevertheless selected, but is clearly less significant. One explanation is that, from the detailed PCA results reported in [4], we can see that *class-level* dynamic coupling measure tend to be more correlated to size and static coupling and, similarly, dynamic *export* coupling measures tend to be less correlated to size measures than their *import* counterpart. A likely reason is that it is easy to imagine small classes providing services to many other methods and, therefore, having a large export coupling. Large import coupling classes though, are more likely to be large because they use many features from other classes.

Results in our earlier study on a Smalltalk system [2] also showed that dynamic export coupling is a stronger indicator of change proneness. Though the context, programming language, and application domain were different, the result obtained in the two studies are consistent, thus suggesting our results can be generalized to a large proportion of systems.

5 Related Work

A large body of work exists on the static measurement of cohesion and coupling, both for procedural [25] and object-oriented systems [15], [24]. In particular, a number of people have used static coupling measurement to assess the maintainability of object-oriented systems [23], [27].

In a number of occasions, those measures have shown to be useful predictors of certain quality attributes such as fault-proneness or change (see survey of empirical results in [7]). For further details on the measures themselves, we refer the reader to surveys that have been published in [8] and [9] where most existing measures and their properties are discussed in detail.

The general idea of using dynamic analysis of programs to assess software quality is not new. For example, Sneed and Merey [26] have shown how it could be used to check assertions and monitor the behavior of modules in procedural software. More specifically, dynamic object-oriented coupling measures were first proposed in [29]. The authors proposed two object-level dynamic coupling measures, Export Object Coupling (EOC) and Import Object Coupling (IOC), based on executable Real-Time Object Oriented Modeling (ROOM) design models. The design model used to collect the coupling measures is a special kind of sequence diagram that allows execution simulation.

IOC and EOC count the number of messages sent between two distinct objects o_i and o_j in a given ROOM sequence diagram x, divided by the *total* number of

TABLE 9
Definition of Size and Static Coupling Measures

Name	Definition
NMI	The number of methods implemented in a class (non-inherited or overriding methods)
NMD	The number of inherited methods in a class, not overridden
NM	The number of all methods (inherited, overriding, and non-inherited) methods of a class. NM = NMI + NMD
NMpub	The number of public methods implemented in a class.
NumPara	Number of parameters. The sum of the number of parameters of the methods implemented in a class.
CS1	The number of source lines of code in a class
CS2	The number of declarations and statements (semicolons) in a class
CBO	Coupling between object classes. According to the definition of this measure, a class is coupled to another, if methods of one class use methods or attributes of the other, or vice versa. CBO is then defined as the number of other classes to which a class is coupled. This includes inheritance-based coupling (coupling between classes related via inheritance).
RFC	Response set for class. The response set of a class consists of the set M of methods of the class, and the set of methods directly or indirectly invoked by methods in M. In other words, the response set is the set of methods that can potentially be executed in response to a message received by an object of that class. RFC is the number of methods in the response set of the class.
RFC_1	Same as RFC, except that methods indirectly invoked by methods in M are not included in the response set.
DAC'	The number of different classes that are used as types of attributes in a class.
PIM_EC	Export coupling version of PIM. The number of invocations of methods of a class c by other classes (regardless of the relationship between classes).
ACAIC OCAIC DCAEC OCAEC ACMIC OCMIC DCMEC OCMEC AMAIC DMAIC AMMIC OMMIC DMMEC OMMEC	These coupling measures are counts of interactions between classes. The measures distinguish the relationship between classes (friendship, inheritance, none), different types of interactions, and the locus of impact of the interaction. The acronyms for the measures indicates what interactions are counted: The first or first two letters indicate the relationship (A: coupling to ancestor classes, D: Descendents, O: Others, i.e., none of the other relationships). The next two letters indicate the type of interaction: CA: There is a Class-Attribute interaction between classes c and d, if c has an attribute of type d. CM: There is a Class-Method interaction between classes c and d, if class c has a method with a parameter of type class d. MM: There is a Method-Method interaction between classes c and d, if c invokes a method of d, or if a method of class *d* is passed as parameter (function pointer) to a method of class *c*. The last two letters indicate the locus of impact: IC: Import coupling, the measure counts for a class c all interactions where c uses another class. EC: Export coupling: count interactions where class d is the used class.

A complete list of measures and appropriate references are provided in [4].

messages in x. Thus, the result is a percentage that reflects the "intensity" of the interaction of two objects related to the total amount of object interaction in x. For example, in a simple scenario $x1$ where o_1 sends two messages (m1 and m2) to o_2 and o_2 sends one message (m3) to o_1, then $\mathrm{IOC}_{\mathrm{x1}}(\mathrm{o}_1, \mathrm{o}_2) = 100^*2/3 = 66\ \mathrm{percent}$ and $\mathrm{IOC}_{\mathrm{x1}}(\mathrm{o}_2, \mathrm{o}_1) = 100^*1/3 = 33\ \mathrm{percent}$. Based on these basic measures, the authors also derive measures at the system level using the probability of executing each sequence diagram as a weighting factor. In a different paper, a methodology for architecture-level risk assessment based on the dynamic measures is proposed [30].

There are several important differences between the measures presented in [29] and the coupling measures described in this paper:

- The dynamic coupling measures in [29] do not adhere to the coupling properties described in the axiomatic framework described in [10]. This is not necessarily a problem in the application context of that particular piece of work, but it would very likely be a problem in many other situations (see [10] for a detailed discussion).
- The measures described in this paper differentiate between many different dimensions of coupling, in addition to import and export coupling. Most importantly, we account for inheritance and polymorphism by distinguishing between dynamic class-level and object-level measures. In our opinion, the ability to measure coupling precisely for systems with inheritance and dynamic binding represents one of the primary advantages of dynamic coupling over static coupling. This is supported by the results presented in the previous section.
- Our measures are collected from analyzing message traces from system executions (Section 3.1) or from UML diagrams (Section 3.2). The dynamic coupling measures in [29] are collected from ROOM models.

Another important addition over [29] is that we perform an empirical validation of our dynamic coupling measures by showing they are complementary to simple size measures and static coupling measures. Furthermore, the relationship of all these measures to an external quality indicator (change proneness) is investigated.

The measures proposed and validated in this paper are based on an initial study described in [2]. Initially, the dynamic coupling measures were described informally, and an initial validation was performed on a SmallTalk system. In this paper, this research has been extended in several important ways. The dynamic coupling measures have been defined formally and precisely in an operational form. As part of this process, we discovered that some of the measures proposed in [2] did not fully adhere to the coupling properties described in [10]. The measures proposed in this paper are shown to be *theoretically* valid, at least based on a widely referenced axiomatic framework. The empirical validation in this paper is also considerably more comprehensive than in [2]. Furthermore, the dynamic coupling measures are compared with size and static coupling measures. Such a comparison was not possible for the SmallTalk system investigated in [2] because static measures could not be collected. This paper clearly confirms the initial empirical evaluation described in [2]; both in terms of Principal Component Analysis and evaluation of the dynamic coupling measures as predictors of change proneness. Thus, the two studies provide a strong body of evidence that the proposed dynamic coupling measures (especially export coupling) are useful indicators of change proneness and capture different properties than do static coupling measures. Results were found to be very similar (despite some differences in measurement) across two separate application domains (commercial CASE tool and open-source web software, respectively) and programming languages (SmallTalk and Java, respectively).

6 Conclusions

The contribution of this paper can be summarized as follows. First, we provide formal, operational definitions of dynamic coupling measures for object-oriented systems. The motivation for those measures is to complement existing measures that are based on static analysis by actually measuring coupling at runtime in the hope of obtaining better decision and prediction models because we account precisely for inheritance, polymorphism and dynamic binding. Second, we describe a tool whose objective is to show how to collect such measures for Java systems effectively and, finally, yet importantly, we perform a thorough empirical investigation using open source software. The objective was three-fold: 1) Demonstrate that dynamic coupling measures are not redundant with static coupling measures, 2) Show that dynamic coupling measures capture different properties than simple size effects, and 3) Investigate whether dynamic coupling measures are useful predictors of change proneness. Admittedly, many other applications of dynamic coupling measures can be envisaged. However, investigating change proneness was used here to gather initial but tangible evidence of the practical interest of such measures.

Our results show that dynamic coupling measures indeed capture different properties than static coupling measures, though some degree of correlation is visible, as expected. Dynamic export coupling measures were shown to be significantly related to change proneness, in addition to that which can be explained by size effects alone. Last, some of the dynamic coupling measures, especially the export coupling measures (EC_OC, EC_OM, EC_OD), appear to be significant (p-value = 0.0001), complementary indicators of change proneness when combined with both size and static coupling measures. The model including dynamic coupling measures yields a R^2 of 0.85, suggesting that a large percentage of variance in code change can be explained by the model. Some of these results confirm those obtained on an earlier study [2] of a SmallTalk system. Though no comparison with static coupling and size measures could be performed in this earlier study, those combined results constitute evidence that dynamic export coupling measures are significant indicators of change proneness.

The results above should be qualified in a number of ways. With respect to external validity, the system we used as a case study may use much more polymorphism and dynamic binding than most systems, thus making dynamic coupling of particular importance. In terms of internal validity, it is clear coupling is only one of the factors affecting change proneness. This is particularly true for requirements changes and recall that our study only considered correction changes. To build complete change proneness models, many other factors would have to be considered. But, this is out of the scope of this paper as the purpose of analyzing change proneness was only to provide an empirical validation of our dynamic coupling measures. Another practical limitation is that using dynamic coupling requires extensive test suites to exercise the system. Such test suites may not be readily available.

Future work will include investigating other applications of dynamic coupling measures (e.g., test case prioritization), and the cost-benefit analysis of using change proneness models such as the ones presented in the current work. These models may be used for various purposes, such as focusing supporting documentation on those parts of a system that are more likely to undergo change, or making use of design patterns to better anticipate change. Note that such applications may also be relevant in procedural software making use of dynamic binding.

Furthermore, a number of other applications of dynamic coupling measurement should be investigated. A side effect of the work presented in this paper is that the JDissect tool can be used to discover dead code, assuming that test data representative of the operational profile of the system is available. Similarly, the tool can be used to determine exactly which objects, classes, and methods are involved in a given functional component (e.g., a use case) of a system. Such functionality could be useful for maintainers to achieve an initial understanding of (complex parts of) a system.

Acknowledgments

The authors would like to thank Magne Jørgensen, Vigdis By Kampenes, Amela Karahasanovic, Dag Sjøberg, Kristin Skoglund, Ray Welland, Jürgen Wüst and the anonymous reviewers for valuable contributions to the research presented in this paper. Lionel Briand was partly funded by an NSERC operational grant and a Canada Research Chair. L.C. Briand is also a visiting professor at the Simula Research Laboratory, Lysaker, Norway.

REFERENCES

[1] E. Arisholm, "Empirical Assessment of Changeability in Object-Oriented Software," PhD Thesis, Dept. of Informatics, Univ. of Oslo, ISSN 1510-7710, 2001.

[2] E. Arisholm, "Dynamic Coupling Measures for Object-Oriented Software," *Proc. Eighth IEEE Symp. Software Metrics (METRICS '02),* pp. 33-42, 2002.

[3] E. Arisholm, D.I.K. Sjøberg, and M. Jørgensen, "Assessing the Changeability of Two Object-Oriented Design Alternatives—A Controlled Experiment," *Empirical Software Eng.,* vol. 6, no. 3, pp. 231-277, 2001.

[4] E. Arisholm, L.C. Briand, and A. Føyen, "Dynamic Coupling Measurement for Object-Oriented Software," Technical Report 2003-05, Simula Research Laboratory, http://www.simula.no/~erika, 2003.

[5] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language Users Guide.* Addison-Wesley, 1998.

[6] L. Bratthall, E. Arisholm, and M. Jørgensen, "Program Understanding Behaviour During Estimation of Enhancement Effort on Small Java Programs," *Proc. Third Int'l Conf. Product Focused Software Process Improvement (PROFES 2001),* 2001.

[7] L.C. Briand and J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems," *Advances in Computers,* vol. 59, pp. 97-166, 2002.

[8] L.C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Software and Systems Modeling,* vol. 1, no. 1, pp. 10-42, 2002.

[9] L.C. Briand, J. Daly, and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.,* vol. 3, no. 1, pp. 65-117, 1998.

[10] L.C. Briand, J.W. Daly, and J. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.,* vol. 25, no. 1, pp. 91-121, 1999.

[11] L.C. Briand, J. Wust, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," *Proc. Int'l Conf. Software Maintenance (ICSM '99),* pp. 475-482, 1999.

[12] F. BritoeAbreu, "The MOOD Metrics Set," *Proc. ECOOP '95 Workshop Metrics,* 1995.

[13] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. Software Systems,* vol. 26, no. 8, pp. 786-796, 2000.

[14] M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, and G. Saint-Denis, "Design Properties and Object-Oriented Software Changeability," *Proc. Fourth Euromicro Working Conf. Software Maintenance and Reeng.,* pp. 45-54, 2000.

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

[16] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Eng.,* vol. 24, no. 8, pp. 629-637, 1998.

[17] I.S. Deligiannis, M. Shepperd, S. Webster, and M. Roumeliotis, "A Review of Experimental Investigations into Object-Oriented Technology," *Empirical Software Eng.,* vol. 7, no. 3, pp. 193-232, 2002.

[18] G. Dunteman, *Principal Component Analysis.* SAGE, 1989.

[19] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Trans. Software Eng.,* vol. 27, no. 7, pp. 630-650, 2001.

[20] R.J. Freund and W.J. Wilson, *Regression Analysis: Statistical Modeling of a Response Variable.* Academic Press, 1998.

[21] Jakarta, "The Apache Jakarta Project," http://jakarta.apache.org/, 2003.

[22] Java.net, "Java Compiler Compiler (JavaCC)," https://javacc.dev.java.net/, 2003.

[23] H. Kabaili, R. Keller, and F. Lustman, "Cohesion as Changeability Indicator in Object-Oriented Systems," *Proc. IEEE Conf. Software Maintenance and Reeng. (CSRM),* pp. 39-46, 2001.

[24] A. Lakhotia and J.-C. Deprez, "Restructuring Functions with Low Cohesion," *Proc. IEEE Working Conf. Reverse Eng. (WCRE),* pp. 36-46, 1999.

[25] G. Myers, *Software Reliability: Principles and Practices.* Wiley, 1976.

[26] H. Sneed and A. Merey, "Automated Software Quality Assurance," *IEEE Trans. Software Eng.,* vol. 11, no. 9, pp. 909-916, 1985.

[27] M.M.T. Thwin and T.-S. Quah, "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics," *Proc. IEEE Int'l Conf. Software Maintenance (ICSM),* 2003.

[28] J. Warmer and A. Kleppe, *The Object Constraint Language.* Addison-Wesley, 1999.

[29] S. Yacoub, H. Ammar, and T. Robinson, "Dynamic Metrics for Object-Oriented Designs," *Proc. IEEE Sixth Int'l Symp. Software Metrics (Metrics '99),* pp. 50-61, 1999.

[30] S. Yacoub, H. Ammar, and T. Robinson, "A Methodology for Architectural-Level Risk Assessment using Dynamic Metrics," *Proc. 11th Int'l Symp. Software Reliability Eng.,* pp. 210-221, 2000.

Erik Arisholm received the MSc degree in electrical engineering from University of Toronto and the PhD degree in computer science from University of Oslo. He has seven years of industry experience in Canada and Norway as a lead engineer and design manager. He is now a researcher in the Department of Software Engineering, Simula Research Laboratory and an associate professor in the Department of Informatics at the University of Oslo. His main research interests are object-oriented design principles and methods, static and dynamic metrics for object-oriented systems, and methods and tools for conducting controlled experiments in software engineering. He is a member of the IEEE.

Lionel C. Briand received the BSc and MSc degrees in geophysics and computer systems engineering from the University of Paris VI, France. He received the PhD degree in computer science, with high honors, from the University of Paris XI, France. He is on the faculty of the Department of Systems and Computer Engineering at Carleton University, Ottawa, Canada, where he founded and leads the Software Quality Engineering Laboratory (http://www.sce.carleton.ca/Squall/Squall.htm). He has been granted the Canada Research Chair in Software Quality Engineering and is also a visiting professor at the Simula Research Laboratory, Lysaker, Norway. Before that, he was the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany. Dr. Briand also worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international, IEEE conferences such as ICSE, ICSM, ISSRE, and METRICS. He is the coeditor-in-chief of *Empirical Software Engineering* (Kluwer), and is a member of the editorial boards of *Systems and Software Modeling* (Springer) and *IEEE Transactions on Software Engineering*. His research interests include: object-oriented analysis and design, inspections and testing in the context of object-oriented development, quality assurance and control, project planning and risk analysis, and technology evaluation. He is a member of the IEEE.

Audun Føyen received the MSc degree in computer science from the University of Oslo in 2004. He has worked as an independent IT consultant for six years. Prior to his stint as a consultant, he worked briefly as a scuba diving instructor in Egypt. His current job involves creating operating systems for mobile phones.