IMPLEMENTATION OF DYNAMIC COUPLING MEASUREMENT OF DISTRIBUTED OBJECT ORIENTED SOFTWARE BASED ON TRACE EVENTS

S.Babu*, R.M.S.Parvathi**

** Research Scalar, Anna University of Technology, Coimbatore.*
***Principal, Sengunthar College of engineering for women, Tiruchengode, Namakkal*
babubalaji2k5@gmail.com
rmsparvathi@india.com

ABSTRACT:
Software metrics are increasingly playing a central role in the planning and control of software development projects. Coupling measures have important applications in software development and maintenance. Existing literature on software metrics is mainly focused on centralized systems, while work in the area of distributed systems, particularly in service-oriented systems, is scarce. Distributed systems with service oriented components are even more heterogeneous networking and execution environment. Traditional coupling measures take into account only "static" couplings. They do not account for "dynamic" couplings due to polymorphism and may significantly underestimate the complexity of software and misjudge the need for code inspection, testing and debugging. This is expected to result in poor predictive accuracy of the quality models in distributed Object Oriented systems that utilize static coupling measurements. In order to overcome these issues, we propose a hybrid model in Distributed Object Oriented Software for measure the coupling dynamically. In the proposed method, there are three steps such as Instrumentation process, Post processing and Coupling measurement. Initially the instrumentation process is done. In this process the instrumented JVM that has been modified to trace method calls. During this process, three trace files are created namely .prf, .clp, .svp. In the second step, the information in these file are merged. At the end of this step, the merged detailed trace of each JVM contains pointers to the merged trace files of the other JVM such that the path of every remote call from the client to the server can be uniquely identified. Finally, the coupling metrics are measured dynamically. The implementation results show that the proposed system will effectively measure the coupling metrics dynamically.

Keywords:
Distributed Object Oriented (DOO) Systems, Software Engineering, Dynamic coupling, Static coupling, Instrumentation, Trace events.

1. INTRODUCTION

Software engineering is an engineering discipline that is concerned with all aspects of software production. Software products consist of developed programs and associated documentation.

Essential product attributes are maintainability, dependability, efficiency and usability. The software process consists of activities that are involved in developing software products. Basic activities are software specification, development, validation and evolution. Methods are organized ways of producing software. They include suggestions for the process to be followed, the notations to be used, rules governing the system descriptions which are produced and design guidelines. Object-oriented technology is built upon a sound engineering foundation, whose elements are collectively called the object model [1]. This model encompasses many useful software construction features such as abstraction, encapsulation, modularity, inheritance, typing, genericity and dynamic binding. Therefore, object model is useful for understanding problems, communicating with application experts and modelling complex enterprises into a software design [2]. This technology also helps to promote software reusability, maintainability, reliability and performance [3].The popularity of the internet coupled with advances in local area network and high speed network technologies have introduced many new distributed applications. Examples include software in the area of computer supported collaborative work, airline and hotel reservation systems, and banking systems - to name just a few. Object-oriented techniques are often used to cope with the complexity of developing these software systems which have come to be known as distributed object-oriented software systems (DOOSS) [4].

Distributed systems have also become increasingly common as more and more organizations use networks to share resources, enhance communication and increase performance. Examples of these systems range from the Internet, to workstations in a local area network within a building, to processors within a single multiprocessor [5]. In a distributed object-oriented application, classes can run on a separate computer within a network system. So, they should be distributed efficiently among different nodes. [6]. A distributed OO application consists mainly of a set of interacting objects; each one runs on a separate computer within a network system. There have been a large number of projects related to the use of object-oriented approaches for the design of problem solving environments for complex applications in various scientific fields [7]. The object model and distributed technologies are being amalgamated [8]. The advantage is obvious: the complexity and dependencies of the entities can make use of the object model in a distributed system to break down the intensive design process into efficient constructs. Many of the concepts of object-oriented programming are currently finding widespread application in loosely coupled distributed systems.

Coupling measures have important applications in software development and maintenance. They are used to help developers, testers and maintainer's reason about software complexity and software quality attributes. The current research on modelling and measuring the relationships among software components through coupling analysis is insufficient. Coupling measures are incomplete in their precision of definition and quantitative computation. In particular, current coupling measures do not reflect the differences in and the connections between design level relationships and implementation level connections. Hence, the way coupling is used to solve problems is not satisfactory. Traditional coupling measures take into account only"static" couplings. They do not account for "dynamic" couplings due to polymorphism and may significantly underestimate the complexity of software and misjudge the need for code inspection, testing and debugging. Due to inheritance, the class of the object sending or receiving a message may be different from the class implementing the corresponding method.

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 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. Static analysis cannot capture all the dimensions of object-level coupling, as features of object-oriented programming such as polymorphism, dynamic binding and inheritance in evaluating the run-time behaviour of an application [9]. As per the previous research and experiments, dynamic coupling is more precise than static coupling for systems with unused code [10].

In the proposed system, we are going to calculate the coupling measures dynamically in a Distributed Object Oriented (DOO) system. Dynamic coupling in DOO system means, calculating the coupling dynamically on both clients and server. The rest of the paper is organized as follows: Section 2 deals with some of the recent research works related to the proposed technique. Section 3 explains the Distributed Object Oriented Systems. Section 4 describes the proposed technique for coupling measurement with all necessary mathematical formulations and figures. Section 5 discusses about the experimentation and evaluation results with necessary tables and graphs and section 6 concludes the paper.

2. DYNAMIC COUPLING MEASUREMENT

2.1 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. 1, 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. 1 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 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.

2.1.1 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.1.

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, C= AC ∪ LC ∪ FC and AC ∩ LC ∩ FC = φ ;. 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).

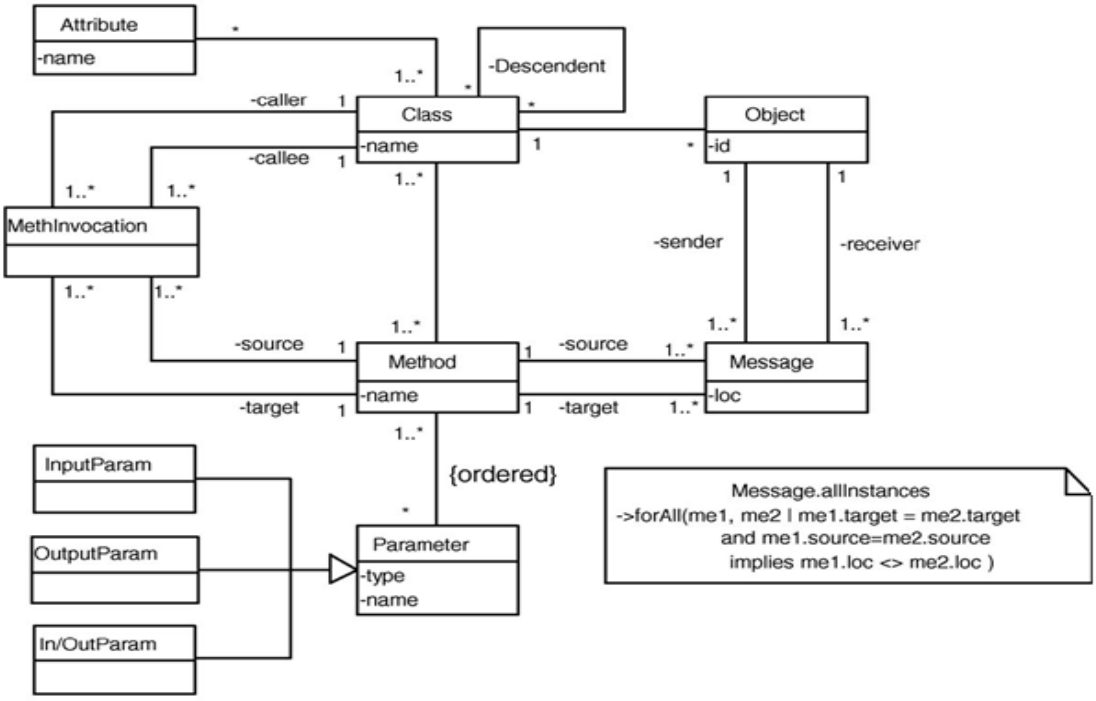

Fig.1. Class diagram capturing a data model of the dynamic analysis information.

2.1.2 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 x 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 x M x N x O x 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. 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 x C x M x 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, RMC $\subseteq$ M x C refers to methods being defined in classes, a binary relation from the set of methods to
the set of classes.

2.1.3 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.

$(\exists(o1c1),(o2c2) \in Roc)\ (\exists 1 \in N)\ (o1m1|o2m2) \in ME \Rightarrow (\exists c3 \in A\ (c1) \cup \{c1\},\ c4 \in A(c2) \cup \{c2\})$
$((m1c3) \in R_{MC} \wedge ((\forall c5 \in A\ (c1) - \{c3\})(m1c5) \in R_{MC} \Rightarrow c5 \in A(c3))) \wedge$
$((m2c4) \in R_{MC}) \wedge ((\forall c6 \in A\ (c2) - \{c4\})\ (m2c6) \in R_{MC} \Rightarrow c6 \in A\ (c4))) \wedge (m1c3, m2c4) \in IV.$

2.1.4 Working Example

We now use a small working example, as shown in Fig. 2, 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 pseudo code. 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.

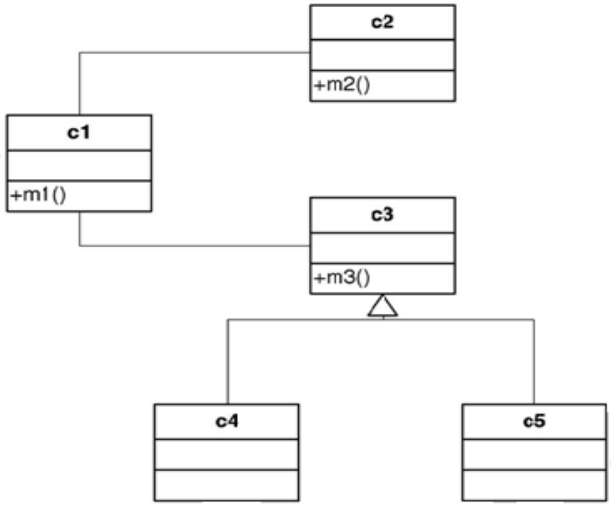

Fig.2. Working class diagram example (UML notation).

The following sets can be derived from Fig. 2:

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. 3, 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., SDi:objectid). Similarly, we denote the line of code of the method invocation in message tuples as l (SDi:messageid). In the example, we assume that the line of code of the method invocations m3ðÞ in messages SD1:1:1, SD1:1:2, and SD1:1:3 are different. Furthermore, since the sequence diagrams do not specify the sender object, source class and source method of the method invocations m1ðÞ in messages SD1:1 and SD2:1, the example sets derived below account for only the four (completely specified) messages SD1:1:1, SD1:1:2, SD1:1:3, and SD2:1:1:

O = {SD1 : 1; SD1 : 2; SD1 : 3; SD2 : 1; SD2 : 2}
R_{OC} = {(SD1 : 1; c1) ; (SD1 : 2; c4) ; (SD1 : 3; c5) ; (SD2 : 1; c1) ; (SD2 : 2; c2)}
ME = {(SD1 : 1; m1; l(SD1 : 1:1) ; SD1 : 2; m3); (SD1 : 1; m1; l(SD1 : 1:2) ; SD1 : 3; m3); (SD1 : 1; m1; l(SD1 : 1:3); SD1 : 3; m3); (SD2 : 1; m1; l(SD2 : 1:1); SD2 : 2; m2)}
IV = {(m1; c1; m3; c3) ; (m1; c1; m2; c2)}:

2.1.5 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 [11], 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 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

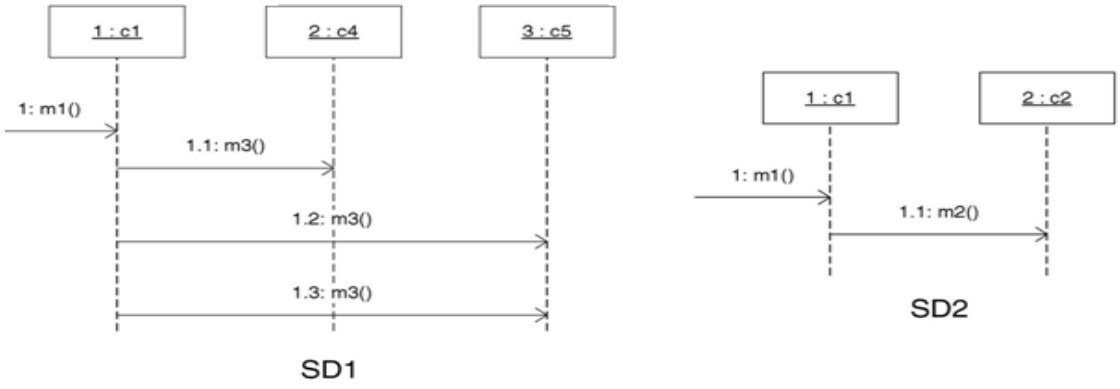

Fig. 3. Two hypothetical sequence diagrams related to Fig. 2.

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

TABLE 1 : Summary of Dynamic Coupling Measures

.

Distinct method invocations. A simpler alternative is to count the number of distinct methods invoked by each method in each object (that invokes 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 1 show 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 (c1≠ c2), because we are focusing on dependencies that contribute to coupling between classes, not their cohesion (as further discussed in [11], [12].Reflexive method invocations are therefore excluded.

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 2: Example Coupling Sets When the Entity of Measurement is the Class

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 3: Example Coupling Sets When the Entity of Measurement is the Object

IC_CD(c1)	{(m1,c1,I(SD$_1$:1.1),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)}

TABLE 4: Changed Import Coupling Sets after Adding a New Implementation of m3() in c5

2.1.6 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, SC ∈ 2^C, and we can define:

$$IC\_CM(SS) = \cup_{(all\ o \in SC)} IC\_CM(c):$$

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

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

Similar definitions can be provided for all levels of granularity.

2.1.7 Example

Returning to our working example in Figs. 2 and 3, 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 2. 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. 2 by adding a new implementation of method m3() in c5: R_{MC} = {(m1; c1); (m3; c3); (m3; c5); (m2; c2)}, while keeping the sequence diagrams in Fig.3 unchanged. This results in a new element in IV :

$$IV = \{(m1; c1; m3; c3); (m1; c1;m3; c5); (m1; c1; m2; c2)\}:$$

The other sets (C, M, O, 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 4, 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 (IC_ Ox(c)) 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 5).

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)
EC_CM(c3)	{(m1,c1,m3,c3)}
EC_CC(c3)	{(m1,c1,c3)}
EC_CD(c5)	**{(m1,c1,I(SD_1:1.2),m3,c5), (m1,c1,I(SD_1:1.3),m3,c5)}**
EC_CM(c5)	**{(m1,c1,m3,c5)}**
EC_CC(c5)	**{(m1,c1,c5)}**

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

2.2 Analysis of Properties

We show here that the five coupling properties presented in [9] 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 below can be performed in a similar way for all coupling measures, at all levels of granularity.

Non Negativity. 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 (m, c, m', c') $\in$ M X C X M X 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:

$$ME = 0 \Leftrightarrow IC\_OM(S) = 0:$$

This is consistent with our intuition as this should be the only case where we get a null coupling value. Since ME = 0 $\Leftrightarrow$ IV = 0 (consistency rule), we also have:

$$ME = 0 \Leftrightarrow IC\_CM(S) = 0$$

At the object level, for IC_OM (o), we have:

$$(\forall o \in O, m \in M, l \in N, o' \in O, m' \in M)$$
$$(o, m, l, o', m') \notin ME \Leftrightarrow IC\_OM(o) = 0.$$

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:

$$(\forall o \in O, c \in C, (o, c) \in Roc)\ IC\_OM(o) = 0$$
$$\Leftrightarrow IC\_CM(c) = 0 \text{ (consistency rule):}$$

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 o $\in$ O sends an additional message (o, m, l, o', m') $\in$ ME, this cannot reduce the number of pairs (method, class) $\in R_{MC}$ that are part of the sets IC_OM(o) or IC_OM(S). The same can be said for export coupling if object o $\in$ 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 c0 is the result of merging c1 and c2, thus transforming system S into S0, for any Coupling measure, we want the following properties to hold at the class and system levels:

$$\text{Coupling (c1)} + \text{Coupling (c2)} \geq \text{Coupling (c0)}$$
$$\text{Coupling (S)} \geq \text{Coupling (S')}:$$

Taking IC_ CD as an example, we can easily show this property holds: All instances of c1 and c2 in IV 's tuples are substituted with c'. If there exist tuples of the type (m1; c1; m2; c2) in IV, then they are transformed into tuples of the form (m1; c'; m2; c'). For IC_Cx measures, since we exclude reflexive method invocations because they do not contribute to coupling then tuples of the form (m1; c'; m2; c') disappear because of the merging. Hence:

$$|IC\_CD\ (c')| \leq |IC\_CD\ (c1)| + |IC\_CD\ (c2)|$$

Similar arguments can be made for all other coupling measures.

3. COLLECTING DYNAMIC COUPLING DATA AT DISTRIBUTED ENVIRONMENT:

Here we propose a hybrid model in Distributed object oriented Software for coupling measurements dynamically. In the proposed method, there are three steps such as Instrumentation process, Post processing and Coupling measurement. Initially the instrumentation process is done. In this process the instrumented JVM that has been modified to trace method calls. During this process, three trace files are created namely .prf, .clp, .svp. In the second step, the information in these file are merged. At the end of this step, the merged detailed trace of each JVMs contains pointers to the merged trace files of the other JVMs such that the path of every remote call from the client to the server can be uniquely identified. Finally, the coupling metrics are measured dynamically. The brief explanation of the proposed method is described as follows

3.1 Introspection Process

There are several different techniques for collecting run-time execution information, including techniques such as sampling and direct instrumentation. Sampling requires the running application to be stopped periodically to obtain information on methods that are currently being executed. The accuracy of the information obtained through sampling is determined by the sampling frequency. A higher sampling frequency can provide more detailed information, but this greater detail comes at the expense of greater perturbation of the executing program. Direct instrumentation, on the other hand, adds code to the JVM to directly measure method execution times. This approach provides more precise information than sampling, since no execution steps are missed. However, the additional instrumentation code may produce greater perturbations than sampling. When humans introspect, they look inside themselves. When our programs introspect, they also look inside themselves, but in a different way.

In object-oriented languages, introspection is the idea that program code can get information on itself. In object-oriented languages, introspection permits programs to get and use information on classes and objects at run-time. Using introspection, one can ask an object for its class, ask a class for its methods and constructors find out the details of those methods and constructors, and tell those methods to execute. Introspection is otherwise known as instrumentation process. The java

class files are executed by instrumented JVM to generate the execution events. Introspection allows the main application to examine each of its plug-ins for the methods it supports and then call them when appropriate. Without introspection, it could not determine the names and parameter types of those methods. To minimize perturbation, the JVM was modified only to the extent necessary to generate enough trace information to visualize the execution call graph. Introspection procedure is:-

1. First we have to compile the source code.
2. Then use Reflection to retrieve data members and methods.

Finally maintain a Vector to store the retrieved information.

3.2 Trace Events

The trace generation module of the JVM is modified to record every invocation of a method using time stamps that show the start and end times of the method with microsecond resolution. As a Java program is executed by the instrumented JVM, three trace files are generated, i.e., .prf, .clp and .svp files.

.prf file

The .prf file contains detailed trace information that records call and return time stamps for every method executed. Invocations of the same method executed under different threads are distinguished from one another by their unique thread identifiers. The other two files record the client/server interaction, if any, that occurs on the JVM as the program is being executed.

.clp file

The .clp file contains information about all of the outgoing RMI calls from the running JVM, i.e., identifying information for remote methods invoked by this JVM.

.svp file

The .svp file records information about all incoming RMI calls, i.e., all of the methods remotely invoked on this JVM by other JVMs. The .clp file is referred to as the client profile of remote methods for which the JVM acts as a client, and the .svp file is referred to as the server profile of remote methods for which the JVM acts as a server. Note that a server JVM may also execute client- type functions and a client JVM may also act as a server to other JVMs.

The trace generation module of the JVM is modified to record every invocation of a method using time stamps that show the start and end times of the method with microsecond resolution. Additionally, a thread identifier is recorded to uniquely identify the thread executing the method. The profiling function creates a new trace record in a buffer for each method entry or exit event. For faster processing, the trace information is stored in main memory and written to external files only when the buffer overflows. Since disk operations are time consuming, it is important to minimize the number of rights to the external file. Since the number of methods called within a program can be quite large, and since each instance of a method generates a trace entry, the amount of trace data generated for a large application would be enormous. Filtering options are provided with the instrumented JVM to reduce the amount of trace data collected. The second

filtering option specifies a list of classes to be traced. For this option, the class hierarchy of the object on which the current method is invoked is checked to see whether it belongs to any of the classes specified. If so, the method is traced. Otherwise, the method invocation is simply ignored.

Client/server traces generation. The Java remote method invocation (RMI) facility allows one JVM to execute a method on another JVM, which may be executing on a physically distributed processor. To match corresponding entries in the server and client profiles, the modified JVM also records the machine (JVM) names. On the client side, the server machine name on which the call was invoked is recorded. On the server side, the client machine name from which the call originated is also recorded. The client-side port number of the TCP/IP connection used for the remote call is recorded in both the server and the client profiles. The port number is needed to distinguish between remote calls made to the same server from different client JVMs residing on the same physical machine. The identifier of the thread that invoked the remote call is recorded on the client side in order to map the detailed trace entry of the remote method invocation to the corresponding client profile entry. Similarly, the identifier of the thread where the remote call is received is recorded on the server side. Finally, the time stamps the time at which the remote call was invoked on the client and the time at which the call was received on the server are also recorded.

3.3 Post processing

The post processing step takes the detailed .prf profile of each JVM, along with the .clp and the .svp files, as input. The merge and tree generation sub steps process these input files to produce a dynamic execution tree for the desired client or server. The details of these steps are described in the following subsections.

3.4 Merge Step

The main part of the merge process is to link client calls recorded in the client profile of a client JVM to the appropriate entry in the server profile of the remote server JVM where the method was actually executed. To see how the corresponding client- and server-side entries are matched, consider a simple scenario where there is one client JVM interacting with one server JVM. In a single-threaded application, multiple calls made from the client to the same remote object and method on the server can easily be matched using remote object and remote method identifiers recorded in the client- and server-profile entries. The entries with the same remote object and remote method identifiers on both sides can be aligned in chronological order and the entries matched in that order. Since the JVM is multithreaded, however, there could be a second call made to the server from the client through another thread before the first call even begins, if Java threads are mapped to different native threads in the JVM implementation. Thus, the second call may arrive at the server before the first one. The remote object and remote method identifiers are not sufficient to unambiguously link the appropriate calls in this case. Consequently, the port number of the TCP/IP connection is used to resolve this ambiguity.

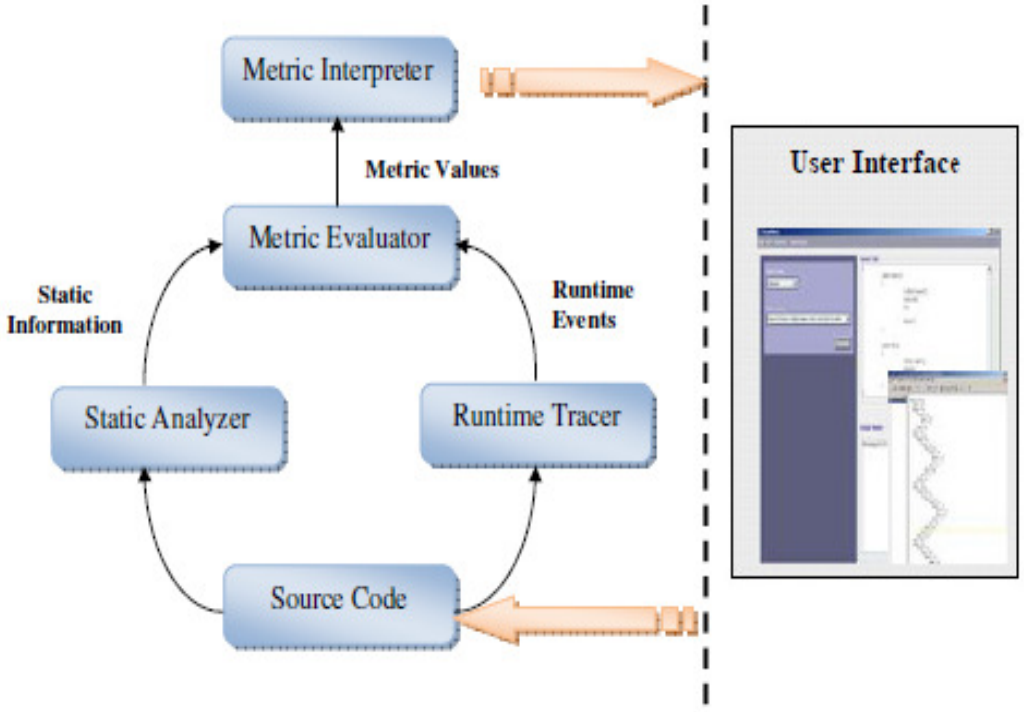

Fig.1: Dynamic Coupling Measurement

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 [13]. 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 sub releases (called "release candidates" in Velocity) within one major release of the Velocity system (version 1.2). The first sub release, 1.2rc1, consists of 17,210 source lines of code (SLOC) in 136 core application classes 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 [14].

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 dynamic coupling tracer tool computed the dynamic coupling measures. Third, size and a comprehensive set of static coupling measures of Velocity. Thus, coupling to/from library and framework classes were not included.

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 [15], 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 [14].

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 sub releases 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. Sub releases in a major release include only correction changes3 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 sub releases 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 [16]. 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 [17].

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 [15], [18]. 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 [14].

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 [14]. 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 was then used to analyze the covariance structure of the measures and determine the underlying dimensions they capture. Detailed results, provided in [14], 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 [14] may not be needed. It is interesting to note that this confirms the PCA results4 in an earlier case study on a Smalltalk system.

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 [14] 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. 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.

5. RELATED WORK

Kai Qian *et al* [19] have presented a service decoupling metrics for service-oriented distributed software composition. The proposed metrics can be applied in the selection of service component in the service-oriented software design process and to evaluate the service-oriented software as a whole in term of decoupling quality attribute for better software understandability, maintainability, reliability, testability, and reusability.

Liguo Yu [20] had presented a method to correlated evolutionary coupling and reference coupling. They studied the evolution of 597 consecutive versions of Linux and measure the evolutionary coupling and reference coupling among 12 kernel modules. They compared 12 pairs of evolutionary coupling data and reference coupling data. The results showed that linear

correlation existed between evolutionary coupling and reference coupling. They concluded that in Linux, the dependencies between software components induced via the system architecture had noticeable effects on kernel module co-evolution.

Pham Thi Quynh *et al*. [21] Service-oriented systems have become popular and presented many advantages in develop and maintain process. The coupling is the most important attribute of services when they are integrated into a system. In this paper, we propose a suite of metrics to evaluate service's quality according to its ability of coupling. We use the coupling metrics to measure the maintainability, reliability, testability, and reusability of services. Our proposed metrics are operated in run-time which bring more exact results.

Byron J. Williams and Jeffrey C. Carver [22] have described in the proposed work a systematic literature review of software architecture change characteristics. The results of the systematic review were used to create the Software Architecture Change Characterization Scheme (SACCS). The proposed report addressed the key areas involved in making changes to software architecture. SACCS's purpose is to identify the characteristics of a software change that would have an impact on the high-level software architecture.

6. CONCLUSION

In this paper, we have proposed a new approach to the computation of dynamic coupling measures in DOO systems by introspection and adding trace events into methods. First, we provide formal, operational definitions of coupling measures and analysis. We propose dynamic coupling measures for distributed object-oriented systems i.e., coupling measurement on both clients and server dynamically. We described the classification of dynamic coupling measures. 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. 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. Finally we propose our dynamic coupling measurement techniques which involve Introspection Procedure, Adding trace events into methods of all classes and Predicting Dynamic Behaviour while running the source code. The source code is filtered to arrive the Actual Runtime used Source Code which is then given for any standard coupling technique to get the Dynamic Coupling.

REFERENCES

[1] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song, "Developing an Object-Oriented Software Testing and Maintenance Environment", Communications of the ACM, Vol. 38, No. 10, pp.75-87, oct.1995.

[2] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenson, "Object-Oriented Modeling and Design", Prentice Hall, 1991.

[3] I .Graham, "Object-Oriented Methods" Addison-Wesley, 1994.

[4] Alan C. Y. Wong, Samuel T chanson, S. C. Cheung and Holger Fuchs, "A Framework for Distributed Object-Oriented Testing", In Proceedings of the IFIP TC6 WG6.1 Joint International Conference on

Formal Description Techniques for Distributed Systems and Communication Protocols (FORTEX) and Protocol Specification, Testing and Verification (PSTV XVII), pp.1-22, 1997.

[5] L. Lamport and N. Lynch, "Distributed Computing: models and methods", Handbook of theoretical Computer science, pp.1157-1199, Elseiver Science, 1990.

[6] Safwat H. Hamad, Reda A. Ammar, Mohammed E. Khalifa and Ayman El-Dessouky, "A Multi step Approach for Restructuring and Mapping Distributed Object-Oriented Software onto a Multiprocessor System", In proc. of the INFOS2008, March 27-29, 2008 Cairo-Egypt, pp.PAR19-PAR23, 2008.

[7] N. Koziris, M. Romesis, P. Tsanakas and G. Papakonstantinou, "An Efficient Algorithm for the Physical Mapping of Clustered Task Graphs onto Multiprocessor Architectures", In Proceedings of the 8th EuroPDP, pp. 406-413, Jan.2000.

[8] R. S. Chin and S. T. Chanson, "Distributed Object-based programming systems", ACM Computing Surveys, Vol.23, No.1, pp.91-124, 1991.

[9] T. W. Ryan, "Distributed object technology: concepts and applications", Prentice Hall, 1997

[10] A. Kavitha and Dr. A. Shanmugam, "Dynamic Coupling Measurement of Object Oriented Software Using Trace Events", In proc. of the 6th International Symposium of Applied Machine Intelligence and Informatics, 2008 (SAMI 2008), pp. 255-259, Jan. 2008.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

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

[18] 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.

[19] Kai Qian, Jigang Liu and Frank Tsui," Decoupling Metrics for Services Composition", In Proc. of the 5th IEEE/ACIS International Conference on Computer and Information Science and 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (ICIS-COMSAR 2006), pp.44-47,2006.

[20] Liguo Yu," Understanding component co-evolution with a study on Linux", Journal of Empirical Software Engineering Archive, Vol. 12, No. 2, pp.123-141, Apr. 2007

[21] Pham Thi Quynh and Huynh Quyet Thang ,"Dynamic Coupling Metrics for Service Oriented Software", International Journal of Electrical and Electronics Engineering, Vol. 3, No.5, pp.282-287, 2009.

[22] Byron J. Williams and Jeffrey C. Carver," Characterizing Software Architecture Changes: A Systematic Review", Information and Software Technology, Vol.52, No.1, pp. 31-51, Jan.2010.