The Conceptual Coupling Metrics for Object-Oriented Systems

Denys Poshyvanyk, Andrian Marcus

Department of Computer Science
Wayne State University
Detroit Michigan 48202
313 577 5408
denys@wayne.edu, amarcus@wayne.edu

Abstract

Coupling in software has been linked with maintainability and existing metrics are used as predictors of external software quality attributes such as fault-proneness, impact analysis, ripple effects of changes, changeability, etc. Many coupling measures for object-oriented (OO) software have been proposed, each of them capturing specific dimensions of coupling.

This paper presents a new set of coupling measures for OO systems – named conceptual coupling, based on the semantic information obtained from the source code, encoded in identifiers and comments. A case study on open source software systems is performed to compare the new measures with existing structural coupling measures. The case study shows that the conceptual coupling captures new dimensions of coupling, which are not captured by existing coupling measures; hence it can be used to complement the existing metrics.

1. Introduction

Many maintenance tasks require the developer to measure directly or indirectly several attributes and assess properties of the software system under evolution. A variety of measures are proposed by researchers to assist developers in getting more complete views of the software.

Coupling is one of the properties with most influence on maintenance as it has a direct effect on maintainability. Proposed coupling measures are used in tasks such as impact analysis [5, 34], assessing the fault-proneness of classes [35], fault prediction [14, 17], re-modularization [1], identifying of software components [22], design patterns [3], assessing software quality [10], etc.

In general, one of the goals of the software designers is to keep the coupling in an OO system as low as possible. Classes of the system that are strongly coupled are most likely to be affected by changes and bugs from other classes; these classes tend to have an increased architectural importance and thus need to be identified. Coupling measures help in such endeavors, and most of them are based on some form of dependency analysis, based on the available source code or design information. The number of dimensions captured by the measures is lower than the number of proposed coupling measures [10], which reflects the fact that many of these measures are based on comparable hypothesis and use similar information for computation.

We proposed a new set of coupling measures, which formulates and captures new dimensions of coupling, i.e., conceptual coupling, based on the semantic information shared between elements of the source code. Our measures can be classified as measuring the strength of conceptual similarities among methods of different classes. The measures are based on using information retrieval (IR) techniques to model and analyze the semantic information embedded in software (i.e., through comments and identifiers).

The conceptual coupling can be used to augment existing measures, especially in tasks such as impact analysis and change propagation, as existing models [5] do not capture all the ripple effects of changes in existing software. They also have direct application in reverse engineering tasks like re-modularization.

The following section outlines the related work for object-oriented coupling metrics. Section 3 describes our approach and the proposed measures. This section also describes implementation details of the tool that we developed to compute our metrics as well as mathematical properties of the measures. In section 4 we provide empirical study to assess the newly proposed metrics. Section 5 concludes the paper and discusses the future work.

2. Related work

Coupling measurement is a very rich and interesting body of research work, resulting in many different approaches using structural coupling metrics [8, 11, 12,

23], dynamic coupling measures [4], evolutionary and logical coupling [16, 38], coupling measures based on information entropy approach [2], coupling metrics for specific types of software applications like knowledge-based systems [20], and more recently systems developed using aspect-oriented approach [37].

The structural coupling metrics have received significant attention in the literature. These metrics are comprehensively described and classified within the unified framework for coupling measurement [6]. The best known among these metrics are CBO (coupling between objects) and CBO^1 [11, 12], RFC (response for class) [11] and RFC_∞ [12], MPC (message passing coupling) [24], DAC (data abstraction coupling) and DAC^1 [24], ICP (information-flow-based coupling) [23], the suite of coupling measures by Briand et al. (IFCAIC, ACAIC, OCAIC, FCAEC, etc) [8]. Other structural metrics like Ce (efferent coupling), Ca (afferent coupling), COF (coupling factor), etc. are also overviewed in [6].

Many of the coupling measures listed above are based on method invocations and attribute references. For example, the RFC, MPC, and ICP measures are based on method invocations only. CBO and COF measures count method invocations and references to both methods and attributes. The suite of measures defined by Briand et al. [8] captures several types of interactions between classes like class-attribute, class-method, as well as method-method interactions. The measures from the suite also differentiate between import and export coupling as well as other types of relationships like friends, ancestors, descendants etc.

Dynamic coupling measures [4] were introduced as the refinement to existing coupling measures due to gaps in addressing polymorphism, dynamic binding, and the presence of unused code by static structural coupling measures.

Another important family of coupling measures derives from the evolution of software system in contrast to structural coupling which is determined by program analysis or dynamic coupling which is obtained by executing the program. These are called evolutionary couplings among parts of the systems which are determined by the common changes or co-changes [38].

Recently, several specialized coupling metrics were proposed for different types of software systems. They are coupling metrics for knowledge-based systems [20] as well as coupling metrics for aspect-oriented programs [37].

Existing work on software clustering [21, 25] uses the concept of semantic similarity between elements of the source code [29], which stands at the foundation of the conceptual coupling, as defined in this paper.

3. Using information retrieval methods for coupling measurement

Our approach to coupling measurement is based on the philosophy that elements (classes) of (OO) software systems are related in more than one way. The obvious and most explored set of relationships is based on data and control dependencies. In addition to such relationships classes are also related conceptually, as they may contribute together to the implementation of a domain concept. We propose here a mechanism to capture and measure this form of coupling, named conceptual coupling.

In our previous work [25, 26] we investigated ways to extract, encode, and analyze the semantic information embedded in the comments and identifiers of the software. Developers use comments and identifiers to represent elements of the problem or solution domain of the software. We use this information to define and identify conceptual coupling between classes.

The underlying mechanism used to extract and analyze the semantic information from the source code is based on Latent Semantic Indexing (LSI) [13], an advanced IR method. We used this mechanism to address other software maintenance tasks such as concept location [30], feature identification [31], identification of abstract data types in legacy source code [25], clone detection [27], and recovery of traceability links between software and documentation [28]. Among other noticeable applications of LSI is software clustering using semantic information [21] which is similarly to our approach based on the assumption that parts of the software system that use similar terms are related.

The definition of the conceptual coupling of classes builds on our previous work on measuring the conceptual cohesion of a class [29]. The source code of the software system is converted into a text corpus where each document contains elements of the implementations of a method. LSI uses this corpus to create a term-by-document matrix, which captures the distribution of terms in methods. Singular Value Decomposition (SVD) is used then to construct a subspace, called the LSI subspace (or semantic space). Each document from the corpus (i.e., method from the source code) is represented as a vector in the LSI subspace. The cosine between two vectors is used as a measure of semantic similarity between two documents (methods). This measure, called conceptual similarity, is used to determine how much relevant semantic information is shared among methods of different classes in the context of the entire system.

Computing conceptual similarities between the methods of two classes indicates whether the classes

are conceptually related. Considering the conceptual similarities of the methods of a class with all the methods of the other classes in the software system, one can measure the degree to which this class relates to the rest of the classes within the context of the software system. These relationships define a new form of coupling, the Conceptual Coupling of Classes (CoCC).

The following subsections define the model and measures we use to asses the CoCC.

3.1. System representation and definitions

Definition 1 (*System, Classes*)

We consider an OO system as a set of classes C = $\{c_1, c_2 \ldots c_n\}$. The number of classes in the system C is n = |C|.

Definition 2 (*Methods of a Class*)

A class has a set of methods. For each class c ∈ C, M(c) = $\{m_1, \ldots, m_z\}$ represent its set of methods, where z = |M(c)| is the number of methods in a class c. The set of all methods in the system is defined as M(C).

Definition 3 (***C**onceptual **S**imilarity between **M**ethods - CSM*)

The *conceptual similarity between methods* $m_k \in M(C)$ and $m_j \in M(C)$, $CSM(m_k, m_j)$, is computed as the cosine between the vectors vm_k and vm_j, corresponding to m_k and m_j in the semantic space constructed by LSI.

$$CSM(m_k, m_j) = \frac{vm_k^T vm_j}{|vm_k|_2 \times |vm_j|_2}$$

As defined, the value of $CSM(m_k, m_j) \in [-1, 1]$, as CSM is simply a cosine in the LSI space. In order to comply with non-negativity property of coupling metrics [6], we refine CSM as:

$$CSM^1(m_k, m_j) = \begin{cases} CSM(m_k, m_j) & if \quad CSM(m_k, m_j) \geq 0 \\ else \quad 0 \end{cases}$$

Definition 4 (***C**onceptual **S**imilarity between a **M**ethod and a **C**lass - CSMC*).

Let $c_k \in C$ and $c_j \in C$ be two distinct ($c_k \neq c_j$) classes in the system. Each class has a set of methods $M(c_k) = \{m_{k1}, \ldots, m_{kr}\}$, where $r = |M(c_k)|$ and $M(c_j) = \{m_{j1}, \ldots, m_{jt}\}$, where $t = |M(c_j)|$. Between every pair of methods (m_k, m_j) there is a similarity measure $CSM(m_k, m_j)$. We define the conceptual similarity between a method m_k and a class c_j as follows:

$$CSMC(m_k, c_j) = \frac{\sum_{q=1}^{t} CSM^1(m_k, m_{jq})}{t}$$, which is the average of the conceptual similarities between method m_k and all the methods from class c_j.

Definition 5 (***C**onceptual **S**imilarity **b**etween two **C**lasses - CSBC*)

We define the conceptual similarity between two classes $c_k \in C$ and $c_j \in C$ as:

$$CSBC(c_k, c_j) = \frac{\sum_{l=1}^{r} CSMC(m_{kl}, c_j)}{r}$$, which is the average of the similarity measures between all unordered pairs of methods from class c_k and class c_j. The definition ensures that the conceptual similarity between two classes is symmetrical, as $CSBC(c_k, c_j) = CSBC(c_j, c_k)$.

3.2. The conceptual coupling of a class

With this system representation we define now a family of measures that approximate the coupling of a class in an OO software system by measuring the degree to which the methods of a class are conceptually related to the methods of other classes.

Definition 6 (***Co**nceptual **C**oupling of a **C**lass - CoCC*)

For a class c ∈ C, conceptual coupling is defined as:

$$CoCC(c) = \frac{\sum_{i=1}^{n} CSBC(c, d_i)}{n-1}$$, where n = |C|, $d_i \in C$, and $c \neq d_i$.

Based on the above definitions, CoCC(c) ∈ [0, 1] ∀ c ∈ C. If a class c ∈ C is strongly coupled to the rest of the classes in the system, then CoCC(c) should be closer to one meaning that the methods in the class are strongly related conceptually with the methods of the other classes. In this case, the class most likely implements concepts that overlap with concepts implemented in other classes (which are related in the context of the software system).

If the methods of the class have low conceptual similarity values with methods of other classes, then the class implements one or more concepts with limited interaction with the rest of the system. The value of CoCC(c) in this case will be close to zero.

In this form, CoCC does not make distinction between method types. If needed, CoCC can be altered to account for overloaded, friend, and other method stereotypes, as discussed in [8].

3.2.1. An example of measuring the conceptual coupling of a class. In order to illustrate how the CoCC metric is computed, let us consider three classes: c1 ={m1, m2}, c2 ={m3, m4, m5} and c3 ={m6, m7, m8} with the conceptual similarities between the methods outlined in Table 1.

In order to compute CoCC for class c1, we need to compute conceptual similarities between classes (c1, c2) and (c1, c3), since CoCC (c1) = (CSBC(c1, c2) + CSBC(c1, c3))/2.

In order to compute the conceptual similarities between c1 and c2, we use the following formula: CSBC(c1, c2) = (CSMC (m1, c2) + CSMC (m2, c2))/2. In this case, CSMC(m1, c2) is an average of conceptual similarities between a method m1 and all other methods in class c2. Thus, CSMC(m1, c2) = (CSM^1(m1, m3) + CSM^1(m1, m4) + CSM^1(m1, m5))/3 = (0.7 + 0.27 + 0.13) / 3 = 0.366. Similarly, CSMC (m2, c2) = (0.68 + 0.34 + 0.25)/3 = 0.423. Therefore, **CSBC(c1, c2)** = (0.366 + 0.423)/2 = **0.3945**.

Analogously, we compute conceptual similarities between classes c1 and c3, **CSBC(c1, c3) = 0.4515**.

Now we are able to compute CoCC(c1), since **CoCC(c1)** = (CSBC(c1, c2) + CSBC(c1, c3))/2 = (0.3945 + 0.4515)/2 = **0.423**. Similarly, **CoCC(c2) = 0.357** and **CoCC(c3) = 0.385.**

Since CoCC is an average measure, we could possibly encounter situations when some pairs of classes are highly related and other are not and the average would not capture those cases.

With that in mind, we refine the CoCC to measure the influence of the highly related pairs of methods in different classes.

Table 1. Conceptual similarities between the methods of the classes c1 (light green), c2 (yellow), and c3 (cyan). Conceptual similarities between methods of the same class (white).

	m1	**m2**	**m3**	**m4**	**m5**	**m6**	**m7**	**m8**
m1	1	0.6	0.7	0.27	0.13	0.3	0.41	0.65
m2	0.6	1	0.68	0.34	0.25	0.41	0.39	0.55
m3	0.7	0.68	1	0.45	0.39	0.56	0.66	0.21
m4	0.27	0.34	0.45	1	0.34	0.47	0.23	0.18
m5	0.13	0.25	0.39	0.34	1	0.05	0.03	0.5
m6	0.3	0.41	0.56	0.47	0.05	1	0.23	0.43
m7	0.41	0.39	0.66	0.23	0.03	0.23	1	0.54
m8	0.65	0.55	0.21	0.18	0.5	0.43	0.54	1

3.2.2. Refining CoCC. In order to capture the influence of highly related methods from different classes, we refine CoCC measure to capture only the strongest method similarities. The goal here is to make sure that our measuring mechanism does not miss classes that are highly coupled even to part of the system, as developers need to be aware of such classes. Thus, we define:

$CSMC_m(m_k, c_j) = max\{CSM^1(m_k, m_{jt}), \forall\ t = 1..|M(c_j)|\}$

The conceptual similarity between method m_k and class c_j is denoted by the highest similarity among all possible pairs of methods between method m_k and all the methods in class c_j.

The conceptual similarity between two classes based on $CSMC_m$ is defined as the following:

$$CSBC_m(c_k, c_j) = \frac{\sum_{l=1}^{r} CSMC_m(m_{kl}, c_j)}{r}$$

The conceptual coupling metrics CoCC for a class c, CoCC, defined using $CSBC_m$ is as the following:

$$CoCC_m(c) = \frac{\sum_{i=1}^{n} CSBC_m(c, d_i)}{n-1}, \text{ where } n=|C|, c \neq d_i.$$

Referring back to the example in the previous subsection, with these new definitions, **$CoCC_m$(c1)** = ($CSBC_m$(c1, c2) + $CSBC_m$(c1, c3))/2 = **0.645**. Similarly, **$CoCC_m$(c2) = 0.486** and **$CoCC_m$(c3) = 0.515.**

Class c1 in our example is the one which has highest values of CoCC and $CoCC_m$ metrics, whereas class c2 has the lowest conceptual coupling. Both metrics support the same results in this case. The purpose of this example was to show how the metrics are computed. Section 4 gives more examples from software systems with some interpretations. Section 4 also shows that $CoCC_m$ complements CoCC, hence both metrics should be used for a more complete assessment of the coupling of a class.

3.3. IRC²M – a measurement tool for CoCC

We have developed a tool for the conceptual coupling measurement, named IRC²M (**I**nformation **R**etrieval based **C**onceptual **C**oupling **M**easurement), to automate the computation of the conceptual coupling measures for C++ programs. IRC²M's indexing component is based on IRiSS [32], an IR-based tool used for source code browsing and exploration.

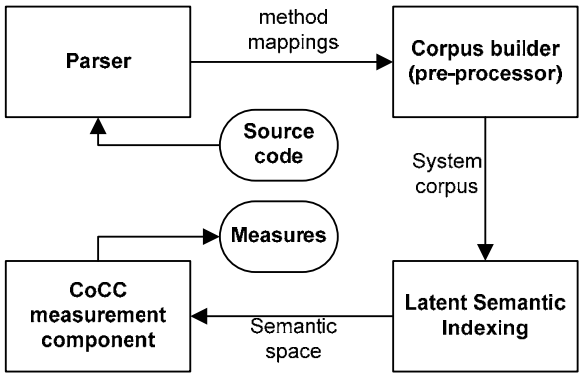

Figure 1. Architecture of the IRC²M tool

IRC²M employs the following steps to compute CoCC (see Figure 1):

- The source code is parsed and the system corpus is constructed by including methods from the source code as documents. For each method in the software system, there will be one document in the corpus. Mappings between methods, classes, and their indexes respectively in the system corpus are generated in this step. Preprocessing of the system corpus is performed to eliminate common keywords, stop words, and to split identifiers [30];
- LSI constructs a term-by-document matrix from the generated system corpus. Then it applies SVD [13] to this matrix to construct the LSI subspace. New document vectors are obtained by orthogonally projecting the corresponding vectors from the original vector space onto the new LSI semantic space.
- Once the methods from the software system are represented in the LSI space, the conceptual similarities between methods are computed (see Def. 3). The CoCC and $CoCC_m$ measures use different measuring mechanisms to determine how the classes are related conceptually in the context of the software system. CSBC and $CSBC_m$ are also computed in this last step.

4. Assessment of the metrics

As CoCC and $CoCC_m$ are new measures, we must evaluate them both theoretically and empirically.

4.1. Theoretical evaluation

We analyze our metrics under the five mathematical properties proposed by Briand et al. [9]: non-negativity, null value, monotonicity, merging of classes, and merging of unconnected classes.

Our measures comply with non-negativity property, based on the redefinition of $CSM^1(m_k, m_j)$ in Def. 3. The null value property is also met since, if similarities of the methods for some class $c \in C$ with the methods of other classes are 0, e.g. $CSM^1(m_k, m_j) = 0$, then the measures have null value, as averages of null values.

While we are not formally proving the later three properties, we are providing the intuition that shows why these properties hold. In short, these properties hold, given that both the mathematical average and the maximum function have these properties. For the monotonicity property, if one adds a new method that has strong conceptual similarities with methods of other classes, then the conceptual coupling measures will also increase. The similar situation will occur if we just change the method implementation which leads to higher conceptual similarities with other methods (e.g. methods share the similar vocabulary). When merging classes and merging unconnected classes, the conceptual similarities remain the same, meaning that relocation of the methods inside other classes will not change actual conceptual similarities of these methods with methods of other classes.

4.2. Case study design

In order to evaluate the proposed coupling measures against existing structural measures, we performed a case study on several open source systems. The goal of the case study is to determine whether the conceptual coupling measures capture new dimensions in coupling measurement.

4.2.1. Coupling metrics

In order to determine whether the newly proposed metrics capture new dimensions in coupling measurement, we selected nine exiting structural metrics for comparison: CBO, RFC, MPC, DAC, ICP, ACAIC, OCAIC, ACMIC, and OCMIC (see Section 2). Originally, we performed a case study with 22 coupling metrics, but we noticed a large amount of redundancy present among the metrics as reported in [10]. The guiding criteria that we used to choose metrics for our case study is the availability of results reported for these metrics in the literature to provide easy comparison and evaluation with our results. For the definitions and explanations of these measures please refer to Section 2.

4.2.2. Subject software systems

For our case study we have chosen ten various sized open-source software systems from different domains. The summary of the selected software systems' sizes are outlined in Table 2. The table also includes specifics on the LSI corpora, generated for the systems under analysis with *terms* standing for the unique number of terms and *docs* for the total number of methods in the software system. The source code for these systems is available at http://sourceforge.net.

ANote (/projects/a-note) is the system that lets the user organize sticky notes on the desktop. TortoiseCVS (/projects/tortoisecvs) is an extension for Microsoft Windows Explorer that makes using CVS convenient and easy. WinMerge (/projects/winmerge) is a tool for visual differencing and merging for both files and directories. Doxygen (/projects/doxygen) is a javaDoc-like documentation system for C++, C, Java, and IDL. Kalpa (/projects/kalpa) is a multi-user client-server accounting, management, CRM, EPR, and MRP system. K-Meleon (/projects/kmeleon) is a fast and customizable Win32 web browser, which uses the same rendering engine as Firefox Mozilla. VoodooUML (/projects/voodoo) is a UML class diagram editor. EMule (/projects/emule) is a file-sharing client; one of the most popular downloads on sourceforge.net. KeePass (/projects/keepass) is a light-weight Win32 password

Table 2. Characteristics of the software systems used in the empirical study. LOC includes lines of code only; the number of lines with comments is provided separately in Comments.

Num	System	Ver	LOC	Comments	Mixed	Files	Classes	Methods	Terms	Docs
1	ANote	4.2.1	16,387	4,731	851	97	61	877	2530	753
2	TortoiseCVS	1.8.21	64,863	15,517	1,541	255	142	930	1915	637
3	WinMerge	2.0.2	51,475	11,534	1,209	169	71	624	1738	522
4	Doxygen	1.3.7	179,920	40,991	8,005	260	682	6837	4424	3608
5	Kalpa	0.0.4.2	16,581	7,330	431	185	135	353	451	254
6	K-Meleon	0.9	34,253	6,940	690	120	57	213	653	192
7	VoodooUML	1.99.12	12,787	2,426	228	97	168	1001	947	841
8	EMule	0.47	162,101	30,542	4,935	556	532	6764	9628	3888
9	KeyPass	1.04	39,798	9,789	1,243	123	104	1476	3676	1325
10	Umbrello	1.5.1	75,665	28,888	1,215	479	210	524	631	405

manager, which allows storing the passwords in a highly-encrypted database. Umbrello (/projects/uml) is a system for creating UML diagrams.

4.2.3. Measurement results

All the structural metrics are collected using the Columbus tool [15] and the conceptual coupling metrics are computed with our IRC^{2}M tool.

We compiled a set of descriptive statistical values (see Table 3): the maximum (max), inter-quartile ranges (25% and 75%), median (med), minimum (min), mean (σ) and standard deviation (μ). The data is used to provide the overall picture of the differences and similarities among all these metrics across the classes from the systems used in the case study.

Table 3. Descriptive statistics for the coupling measures of 979 classes in the 10 software systems

Measures	Min	Max	25%	Med	75%	σ	μ
CoCC	**0.0**	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**
CoCCm	**0.0**	**0.7**	**0.2**	**0.3**	**0.4**	**0.3**	**0.1**
CBO	0.0	45.0	0.0	1.0	4.0	2.9	5.0
RFC	0.0	545.0	5.0	11.0	23.0	24.3	45.1
MPC	0.0	2238	0.0	3.0	13.0	27.2	117
DAC	0.0	59.0	0.0	0.0	2.0	1.4	3.9
ICP	0.0	2779	0.0	2.0	16.0	36.4	162
ACAIC	0.0	1.0	0.0	0.0	0.0	0.1	0.3
OCAIC	0.0	59.0	0.0	0.0	1.0	1.3	3.9
ACMIC	0.0	6.0	0.0	0.0	0.0	0.2	0.5
OCMIC	0.0	88.0	0.0	0.0	2.0	1.8	5.6

4.3. Principal component analysis

In order to understand the underlying, orthogonal dimensions captured by the coupling measures we perform Principal Component Analysis (PCA) on the metrics measured for the software systems in Table 2. We also compare the results of our analysis with those reported in the literature.

4.3.1. Analysis procedure

Briand et al. [10] proposed a methodology to analyze software engineering data in order to make an experiment repeatable and the results comparable. The methodology consists of the following three steps: collecting the data, identifying outliers, and performing PCA.

Section 4.2.3 presented the data we collected. As the results of our analysis can be impacted by the outliers, they were removed. To identify outliers in the data, we utilized the T^{2}max procedure based on the Mahalanobis distance [18].

After outliers were eliminated, we performed PCA, which was used in our case to identify groups of variables (i.e., metrics), which are likely to measure the same underlying dimension (i.e., mechanism that defines coupling) of the object to be measured (i.e., coupling of a classes). In order to identify these variables and interpret the principal components, we consider the rotated components, which is a technique where principal components are subjected to an orthogonal rotation. Thus, the resulting rotated components show clearer patterns of loading for the variables. In order to perform this rotation we used the rotation technique known as "varimax" [19].

4.3.2. PCA results

We performed PCA on the set of 979 classes from 10 different open source software systems (see Table 2). All eleven measures were subjected to an orthogonal rotation. We identified eight orthogonal dimensions spanned by 11 coupling measures. The eight principal components (PCs) capture 97.6% of the variance in the data set, which is significant enough to support our findings.

The loadings on each measure in each rotated component in presented in Table 4. Values higher than 0.5 are highlighted as the corresponding measures are the ones we look into while interpreting the PCs. For every PC, we provide the variance of the data set

Table 4. Rotated components.

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8
Proportion	19.3%	**9.4%**	9.41%	12.1%	19.5%	**9.45%**	9.05%	9.14%
Cumulative	19.3%	28.8%	38.2%	50.4%	69.9%	79.4%	88.4%	97.6%
CoCC	-0.046	**0.941**	0.042	0.000	-0.031	0.279	0.129	-0.033
CoCCm	0.064	0.343	-0.101	0.024	0.115	**0.904**	0.041	0.074
CBO	0.260	-0.147	0.185	**0.558**	0.309	0.341	0.017	0.473
RFC	0.264	-0.019	0.046	0.266	0.422	0.067	0.075	**0.803**
MPC	0.233	-0.017	-0.029	0.154	**0.929**	0.081	0.024	0.202
DAC	**0.931**	-0.027	0.074	0.161	0.268	0.043	0.084	0.136
ICP	0.346	-0.014	-0.024	0.139	**0.903**	0.074	0.028	0.162
ACAIC	0.052	0.035	**0.950**	0.022	-0.040	-0.081	0.272	0.046
OCAIC	**0.935**	-0.026	0.006	0.162	0.274	0.046	0.067	0.127
ACMIC	0.113	0.129	0.281	0.050	0.040	0.041	**0.939**	0.049
OCMIC	0.222	0.029	-0.007	**0.928**	0.181	-0.018	0.052	0.157

explained by the PC and the cumulative variance in Table 4.

Based on our analysis of the coefficients associated with every coupling measure within each of the rotated components, we interpret PCs as the following.

PC1 (19.3%): both DAC and OCAIC measure import coupling from library classes through aggregation. PC2 (9.4%): CoCC measures conceptual coupling of classes within the context of the complete software system. PC3 (9.41%): ACAIC measures import coupling from non-library classes through aggregation. PC4 (12.1%): CBO, OCMIC count import coupling from non-library classes through method invocations. PC5 (19.5%): MPC, ICP have the similar interpretation as the previous component. PC6 (9.45%): $CoCC_m$ measures conceptual coupling of classes based on strongest conceptual similarities among methods of classes in the system. PC7 (9.05%): ACMIC defines a dimension on its own capturing class-method interactions with class types as a parameter. PC8 (9.14%): RFC captures coupling, based on method invocations.

The results of the PCA show that the CoCC and $CoCC_m$ measures define two new dimensions on their own since CoCC is the only significant factor in PC2 and $CoCC_m$ is the only significant factor in PC6. These results clearly indicate that our coupling measures capture different types of coupling between classes, than those captured by the structural metrics. This unique result derives from the fact that CoCC and $CoCC_m$ are coupling measures that are based on completely different ideas and measurements than the existing coupling measures; CoCC and $CoCC_m$ are based on the semantic information obtained from the source code encoded in identifiers and comments, whereas the existing metrics use the structure of the software as the basis for measurement.

In addition, the results of the PCA can be compared with those reported in the literature [7, 10]. Although the PCs and loadings obtained in our case and those reported in the literature do not completely overlap, they are very close. This can be explained by the fact that we used a slightly different set of coupling metrics in our analysis as well as two new metrics, i.e., CoCC and $CoCC_m$.

4.4. Interpretation of the results

To obtain more insight into how the conceptual coupling metrics differ from the structural ones, we chose several classes from different systems for detailed analysis. As the cases where the two sets of metrics agree are of little interest, we were interested in those cases with different values of conceptual and structural metrics, e.g., high conceptual metric values and low structural metrics values, and vice versa. We considered both the CoCC and $CoCC_m$ measures that capture the coupling of the classes to the rest of the system, and the CSBC and $CSBC_m$ measures, which capture conceptual coupling between pair of classes.

4.4.1. Conceptual coupling in the context of the complete software system

In this subsection we look into some of the noted differences between the conceptual coupling measures (CoCC and $CoCC_m$) and the CBO and RFC structural coupling measures.

Table 5. Classes with highest conceptual coupling in WinMerge (W) and TortoiseCVS (T) according to CoCC and $CoCC_m$

S	Class	CoCC	$CoCC_m$	CBO	RFC
W	IVSSItems	0.215	0.326	0	5
W	IVSSUsers	0.215	0.326	0	5
W	IVSSCheckouts	0.215	0.326	0	5
T	ConfListDialog	0.106	0.176	1	5
T	ConflictParser	0.07	0.135	0	1

The chosen classes for detailed analysis are from the WinMerge and TortoiseCVS systems (Table 5). We selected these classes based on high values of CoCC and $CoCC_m$ and low values of CBO and RFC metrics.

The IVSSItems, IVSSUsers, and IVSSCheckouts classes from WinMerge show high conceptual and low structural coupling to the rest of the system. Closer inspection of these classes revealed that these classes are part of a larger cluster of related classes, which contribute to the implementation of a feature related to accessing functions of other ActiveX objects; they all implement the COleDispatchDriver interface. All the classes in the cluster have several common characteristics – they are all wrappers; the majority of the methods in these classes call the InvokeHelper() function to execute specific functionality in the ActiveX object; the majority of pairs of classes from the cluster have high conceptual similarities. The "IVSS" cluster consists of eleven classes wrapping similar functionalities. This explains the high values for CoCC and $CoCC_m$ since these classes are conceptually related to the other classes from the cluster, as well as other classes in the system. Their construction as wrappers and their main usage explains the low structural cohesion.

The classes ConflictParser and ConflictListDialog from the TortoiseCVS system implement important domain concepts - identifying conflicts in the working version of the file and current file revision as well as dialog to list the conflicts in the file. These concepts are important in the system, which extends the file system's interface to support collaborative software development with CVS. The high values of CoCC and $CoCC_m$ metrics for these classes from TortoiseCVS can be explained by the fact that these classes use domain concept terms like "parse" and "conflict", which are spread across many methods of this system. These terms have high global frequencies, meaning that they frequently occur as parts of identifiers or comments across different methods in the system compared to other 1,915 unique terms indexed in this system. The terms "conflict" and "parse" occur more than a thousand times in 679 methods of TortoiseCVS system.

The classes analyzed in this section implement domain concepts, which relate to the rest of the system, yet they are loosely coupled to the rest of the system. It is important to identify these classes from maintenance point of view. The loose structural coupling may indicate a low architectural importance, but the high conceptual coupling indicates that these classes are most likely contributing to the implementation of the main domain concepts. The classes which relate conceptually to the majority of classes in the system may exhibit a form of dependency, called hidden dependency [36], which is not always expressed by structural coupling measures. Modifications in these classes may trigger special types of ripple effects, which are currently not captured by existing coupling measures [5].

4.4.2. Conceptual coupling between a pair of classes

To better understand the conceptual coupling, we also analyze the CSBC and $CSBC_m$ measures, which reflect how a class is conceptually related to another given class in the system.

In order to identify pairs of classes with high conceptual similarity, we computed CSBC and $CSBC_m$ metrics for every possible pair of classes in WinMerge and TortoiseCVS systems. We selected these two systems, since we are mostly familiar with these two systems, among the ten ones used in the case study.

For analysis, we chose several pairs of classes with highest CSBC values (see Table 6).

It came to no surprise that pairs of classes, mentioned before as part of the "IVSS" cluster, were among those with highest CSBC values. These classes implement different, but related tasks, which are all based on implementation of client side of OLE automation. Detailed inspection of the source code for these classes has shown that they are not directly connected structurally, meaning that they do not use each other services etc. On the other hand after inspecting the history of co-changes for these files (using CVS data for WinMerge project) we noticed that these classes are not only strongly conceptually coupled together, but they also have history of common changes (i.e., they were changed and submitted to the repository at the same time).

Table 6. Pairs of classes from WinMerge (W) and TortoiseCVS (T) with highest CSBC values

S	Class	Class	CSBC	$CSBC_m$
W	IVSSVersion	IVSSCheckout	0.776	0.964
W	IVSSItems	IVSSUsers	0.77	0.974
W	IVSSDatabase	IVSSCheckout	0.585	0.954
T	MergeDlg	UpdateDlg	0.375	0.891

Another pair of classes MergeDlg and UpdateDlg from TortoiseCVS system has high conceptual coupling values for CSBC and $CSBC_m$ metrics. This is once again not surprising, since both classes implement

similar concepts – front end dialogs for merging and updating file revisions. Both classes share similar terms which come from names of classes used to create elements of user interface: “button”, “static text”, “check box”, etc., as well as terms more specific to the concepts which are implemented in these classes: “fetch”, “revision”, “tag”, “branch”, etc. Again these classes do not have direct structural dependencies between them. This is a case of unconnected classes, which implement similar functionality [27].

4.5. Threats to validity

We identify several issues that affected the results of our case study and limit our interpretations. We have demonstrated that our metrics capture new dimensions in coupling measurement; however, we obtained these results by analyzing classes from only ten C++ open-source systems. In order to allow for generalization of results, large-scale evaluation, similar to the one in [33] in terms of case study design, is necessary, which will take into account systems from different domains, developed using different programming languages.

In the case study we consider only structural metrics that are based on the static information obtained from the source code. The results may be somewhat different if we considered dynamic coupling [4].

We did not investigate the relationship of the measures to an external quality attributes, e.g. change proneness; such a relationship would be useful to show additional values of the proposed metrics.

5. Conclusions and future work

The paper defines a new set of operational measures for the conceptual coupling of classes, which are theoretically valid and empirically studied. An extensive case study shows that these metrics capture new dimensions in coupling measurement, compared to existing structural metrics.

The paper lays the foundation for a wealth of work that makes use of the conceptual coupling metrics. The proposed metrics could be further extended and refined, for example by taking into account inheritance in measurement. The IRC^{2}M tool will be adapted to compute conceptual coupling measures for Java systems. We are also planning on comparing the conceptual coupling metrics with the evolutionary based coupling [38].

More importantly, we are investigating the applications of the conceptual coupling in impact analysis, detecting hidden dependencies, and change proneness. In addition, we will use these metrics to extend prior work on software clustering [21], concept location [30], and clone detection [27].

6. Acknowledgements

This research was supported in part by grants from the National Science Foundation (CCF-0438970) and the National Institute for Health (NHGRI 1R01HG003491). We thank Prof. Václav Rajlich for his valuable comments and discussions on this research.

7. References

[1] Abreu, F. B., Pereira, G., and Sousa, P., "A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems", in Proc. of Conf. on Software Maintenance and Reengineering (CSMR'00), Zurich, Switzerland, Feb. 29 - Mar. 3 2000, pp. 13-22.

[2] Allen, E. B., Khoshgoftaar, T. M., and Chen, Y., "Measuring coupling and cohesion of software modules: an information-theory approach", in Proceedings of 7th International Software Metrics Symposium (METRICS'01), April 4-6 2001, pp. 124-134.

[3] Antoniol, G., Fiutem, R., and Cristoforetti, L., "Using Metrics to Identify Design Patterns in Object-Oriented Software", in Proceedings of 5th IEEE International Symposium on Software Metrics (METRICS'98), Bethesda, MD, November 20-21 1998, pp. 23 - 34.

[4] Arisholm, E., Briand, L. C., and Foyen, A., "Dynamic coupling measurement for object-oriented software", *IEEE Transactions on Software Engineering*, vol. 30, no. 8, August 2004, pp. 491-506.

[5] Briand, L., Wust, J., and Louinis, H., "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", in Proc. of IEEE International Conf. on Software Maintenance, Aug. 30 - Sept. 03 1999, pp. 475-482.

[6] Briand, L. C., Daly, J., and Wüst, J., "A Unified Framework for Coupling Measurement in Object Oriented Systems", *IEEE Transactions on Software Engineering*, vol. 25, no. 1, January 1999, pp. 91-121.

[7] Briand, L. C., Daly, J. W., Porter, V., and Wüst, J., "A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems", in Proc. of 5th International Software Metrics Symposium (METRICS'98), Bethesda, MD, Nov. 20-21 1998, pp. 43-53.

[8] Briand, L. C., Devanbu, P., and Melo, W. L., "An investigation into coupling measures for C++", in Proc. of International Conference on Software engineering (ICSE'97), Boston, MA, May 17-23 1997, pp. 412 - 421.

[9] Briand, L. C., Morasca, S., and Basili, V. R., "Property-Based Software Engineering Measurements", *IEEE Transactions on Software Engineering*, vol. 22, no. 1, January 1996, pp. 68-85.

[10] Briand, L. C., Wüst, J., Daly, J. W., and Porter, V. D., "Exploring the relationship between design measures and software quality in object-oriented systems", *Journal of Systems and Software*, vol. 51, no. 3, May 2000, pp. 245-273.

[11] Chidamber, S. R. and Kemerer, C. F., "Towards a Metrics Suite for Object Oriented Design", in Proceedings of OOPSLA'91, 1991, pp. 197-211.

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

[13] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", *Journal of the American Society for Information Science*, vol. 41, 1990, pp. 391-407.

[14] El-Emam, K. and Melo, K., "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", *NRC/ERB-1064*, vol. NRC 43609, November 1999.

[15] Ferenc, R., Siket, I., and Gyimóthy, T., "Extracting facts from open source software", in Proc. of 20th International Conf. on Software Maintenance, Sept. 11 2004, pp. 60-69.

[16] Gall, H., Jazayeri, M., Krajewski, J., "CVS Release History Data for Detecting Logical Couplings", *6th International Workshop on Principles of Software Evolution (IWPSE'03)* Sept. 1 - 2, 2003, pp. 13 - 23.

[17] Gyimóthy, T., Ferenc, R., and Siket, I., "Empirical validation of object-oriented metrics on open source software for fault prediction", *IEEE Trans. on Software Engineering*, vol. 31, no. 10, October 2005, pp. 897-910.

[18] Jobson, J. D., *Applied Multivariable Data Analysis*, Springer-Varlag, 1992.

[19] Jolliffe, I. T., *Principal Component Analysis*, Springer Verlag, 1986.

[20] Kramer, S. and Kaindl, H., "Coupling and cohesion metrics for knowledge-based systems using frames and rules", *ACM Trans. on Soft. Engineering and Methodology (TOSEM)*, vol. 13, no. 3, July 2004, pp. 332-358.

[21] Kuhn, A., Ducasse, S., and Girba, T., "Enriching Reverse Engineering with Semantic Clustering", in Proc. of 12th Working Conference on Reverse Engineering, Nov. 7-11 2005, pp. 133-142.

[22] Lee, J. K., Jung, S. J., Kim, S. D., Jang, W. H., and Ham, D. H., "Component identification method with coupling and cohesion", in Proc. of 8th Asia-Pacific Software Engineering Conference (APSEC'01), Dec. 2001, pp. 79-86.

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

[24] Li, W. and Henry, S., "Object-oriented metrics that predict maintainability", *Journal of Systems and Software*, vol. 23, no. 2, 1993, pp. 111-122.

[25] Maletic, J. I. and Marcus, A., "Supporting Program Comprehension Using Semantic and Structural Information", in Proceedings of 23rd International Conference on Software Engineering, Toronto, Canada, May 12-19 2001, pp. 103-112.

[26] Marcus, A., "Semantic Driven Program Analysis", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, IL, September 11-17 2004, pp. 496-473.

[27] Marcus, A. and Maletic, J. I., "Identification of High-Level Concept Clones in Source Code", in Proceedings of Automated Software Engineering (ASE'01), San Diego, CA, November 26-29 2001, pp. 107-114.

[28] Marcus, A., Maletic, J. I., and Sergeyev, A., "Recovery of Traceability Links Between Software Documentation and Source Code", *Int. Journal of Software Engineering and Knowledge Eng.*, vol. 15, no. 4, Oct. 2005, pp. 811-836.

[29] Marcus, A. and Poshyvanyk, D., "The Conceptual Cohesion of Classes", in Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, Sept. 2005, pp. 133-142.

[30] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., "An Information Retrieval Approach to Concept Location in Source Code", in Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, November 9-12 2004, pp. 214-223.

[31] Poshyvanyk, D., Gueheneuc, Y., Marcus, A., Antoniol, G., and Rajlich, V., "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification", in Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC'06), Athens, Greece, 2006, pp. 137-148.

[32] Poshyvanyk, D., Marcus, A., Dong, Y., and Sergeyev, A., "IRiSS - A Source Code Exploration Tool", in Industrial and Tool Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, September 25-30 2005, pp. 69-72.

[33] Succi, G., Pedrycz, W., Djokic, S., Zuliani, P., and Russo, B., "An Empirical Exploration of the Distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite", *Empirical Software Eng.*, 10 (1), Jan. 2005, pp. 81-104.

[34] Wilkie, F. G. and Kitchenham, B. A., "Coupling measures and change ripples in C++ application software", *The Journal of Syst. and Software*, vol. 52, 2000, pp. 157-164.

[35] Yu, P., Systa, T., and Muller, H., "Predicting fault-proneness using OO metrics. An industrial case study", in Proc. of 6th European Conf. on Software Maintenance and Reengineering (CSMR'02), March 2002, pp. 99-107.

[36] Yu, Z. and Rajlich, V., "Hidden Dependencies in Program Comprehension and Change Propagation", in Proc. of 9th Int. Workshop on Program Comprehension, Toronto, Canada, May 12 -13, 2001, pp. 293-299.

[37] Zhao, J., "Measuring Coupling in Aspect-Oriented Systems", in Proc. of 10th IEEE International Soft. Metrics Symposium (METRICS'04), Chicago, USA, 2004.

[38] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", *IEEE Transactions on Software Engineering*, vol. 31, no. 6, June 2005, pp. 429-445.