Empir Software Eng (2009) 14:5–32
DOI 10.1007/s10664-008-9088-2

Using information retrieval based coupling measures for impact analysis

Denys Poshyvanyk · Andrian Marcus · Rudolf Ferenc · Tibor Gyimóthy

Published online: 20 September 2008

Guest Editors: Tim Menzies and Letha Etzkorn

Abstract Coupling is an important property of software systems, which directly impacts program comprehension. In addition, the strength of coupling measured between modules in software is often used as a predictor of external software quality attributes such as changeability, ripple effects of changes and fault-proneness. This paper presents a new set of coupling measures for Object-Oriented (OO) software systems measuring conceptual coupling of classes. Conceptual coupling is based on measuring the degree to which the identifiers and comments from different classes relate to each other. This type of relationship, called conceptual coupling, is measured through the use of Information Retrieval (IR) techniques. The proposed measures are different from existing coupling measures and they capture new dimensions of coupling, which are not captured by the existing coupling measures. The paper investigates the use of the conceptual coupling measures during change impact analysis. The paper reports the findings of a case study in the source code of the Mozilla web browser, where the conceptual coupling metrics were compared to nine existing structural coupling metrics and proved to be better predictors for classes impacted by changes.

Denys Poshyvanyk performed this work while at Wayne State University.

D. Poshyvanyk
Computer Science Department, The College of William and Mary, Williamsburg, VA 23185, USA
e-mail: dposhyvanyk@wm.edu

A. Marcus (✉)
Department of Computer Science, Wayne State University, Detroit, MI 48202, USA
e-mail: amarcus@wayne.edu

R. Ferenc · T. Gyimóthy
Department of Software Engineering, University of Szeged, Szeged, Hungary

R. Ferenc
e-mail: ferenc@inf.u-szeged.hu

T. Gyimóthy
e-mail: gyimi@inf.u-szeged.hu

Keywords Impact analysis · Latent semantic indexing · Information retrieval · Change prediction · Coupling measurement

1 Introduction

During program comprehension, developers need to understand how software modules relate to each other. It is especially important when changes are being made to the software and developers need to assess the impact of their changes. One way to understand such relationships is to measure the coupling between parts of the software. Coupling is one of the fundamental properties of software with a strong influence on comprehension and maintenance of large software systems. Proposed coupling measures are used in software engineering tasks, such as change impact analysis (Briand et al. 1999a; Wilkie and Kitchenham 2000), assessing the fault-proneness of classes (El-Emam and Melo 1999; Yu et al. 2002; Gyimóthy et al. 2005; Olague et al. 2007), software re-modularization (Abreu et al. 2000; Yang et al. 2005), identifying software components (Lee et al. 2001) and design patterns (Antoniol et al. 1998), assessing software quality (Briand et al. 2000), etc.

Depending on the programming paradigm used, the choice of programming language for the implementation, and the design of a software system, coupling is influenced by several factors—such as control and data flow—and hence it may be measured differently. Researchers proposed a variety of coupling measures, but recent studies (Briand et al. 2000) suggest that some of these metrics tend to compute the same form of coupling, though through different measuring mechanisms.

In this work we define a set of coupling measures, which capture new dimensions of coupling, based on the textual information shared between modules of the source code. While elements of the source code written in a programming language help identify control or data flow between software modules, the comments and identifiers express the intent of the software. Two parts of the software with similar intent will most likely refer to the same (or related) concepts in the problem or solution domains of the system. Hence, they are conceptually related. This has been also confirmed by the earlier work of other researchers who examined overlap of semantic information in comments and identifiers among different software modules (Etzkorn and Delugach 2000; Stein et al. 2004). This relationship is the foundation for the new coupling measures, named *conceptual coupling*. The measures are computed using IR techniques that help extract and analyze the textual information embedded in software (i.e., in the comments and identifiers). While any of several IR techniques could be used, in this work, we use Latent Semantic Indexing (LSI) (Deerwester et al. 1990). The set of conceptual coupling metrics can be defined and used for any type of programming paradigm, but we define and use them here in the context of OO software systems.

Existing coupling measures have been previously used to support the impact analysis process, where the task is to identify all classes that would change when a given class is being changed. Existing models (Briand et al. 1999a) do not capture all the ripple effects of changes in existing software. Given that the conceptual coupling metrics reflect different relationships than structural coupling metrics, we assume that they also propagate changes in software. The paper focuses on the use of the conceptual coupling metrics to predict classes that will change during impact analysis. We conducted a case study on a large open-source software system (i.e., Mozilla[1]) to see how the conceptual coupling metrics compare

[1] Mozilla is a web browser and is available at http://www.mozilla.org/ (verified 27/06/08)

with nine existing structural coupling metrics, when used during impact analysis. The case study indicates that one of our conceptual coupling metrics provides best results for predicting classes that need to be changed.

2 Related Work

We are discussing here the major approaches to coupling measurement, in order to contrast between existing approaches and our proposed metrics. The conceptual coupling metrics are based on the use of IR methods and constitute a novel application, compared to previous uses of IR in program comprehension, which we also present here. Coupling measures have been used to support impact analysis and we present those approaches here as well.

2.1 Coupling Measurement

Coupling measurement is a rich and interesting body of research work, resulting in many different measuring approaches for structural coupling metrics (Chidamber and Kemerer 1991; Chidamber and Kemerer 1994; Lee et al. 1995; Briand et al. 1997), dynamic coupling measures (Arisholm et al. 2004; Hassoun et al. 2004), evolutionary and logical coupling (Gall 2003), coupling measures based on information entropy (Allen et al. 2001), coupling metrics for specific types of software applications such as procedural systems (Offutt et al. 1993), knowledge-based systems (Kramer and Kaindl 2004), ontology-based systems (Orme et al. 2006) and systems developed using an aspect-oriented approach (Zhao 2004).

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 (Briand et al. 1999b). The best known among these metrics are CBO (coupling between objects) and CBO' (Chidamber and Kemerer 1991; 1994), RFC (response for class) (Chidamber and Kemerer 1991) and RFC_{∞} (Chidamber and Kemerer 1994), MPC (message passing coupling) (Li and Henry 1993), DAC (data abstraction coupling) and DAC[1] (Li and Henry 1993), ICP (information-flow-based coupling) (Lee et al. 1995), the suite of coupling measures by Briand et al. (Briand et al. 1997) (IFCAIC, ACAIC, OCAIC, FCAEC, etc). Other structural metrics such as C_e (efferent coupling), C_a (afferent coupling) and COF (coupling factor) are also overviewed by Briand et al. (Briand et al. 1999b).

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. (Briand et al. 1997) captures several types of interactions between classes such as class–attribute, class–method, and method–method interactions. The measures from the suite also differentiate between import and export coupling as well as other types of relationships including friends, ancestors, descendants etc.

Dynamic coupling measures (Arisholm et al. 2004; Hassoun et al. 2004) were introduced as the refinement to existing coupling measures due to some 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 of a

single version of software 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 past common changes or co-changes (Gall 2003).

Another form of coupling, namely interaction coupling, captures relations among software artifacts which are relevant to a particular software engineering task (Zou et al. 2007). Interaction coupling uses information gleaned using an Integrated Development Environment on when artifacts are being used or modified in the same development task.

Recently, several specialized coupling metrics were proposed for different types of software systems. They are coupling metrics for knowledge-based systems (Kramer and Kaindl 2004) as well as coupling metrics for aspect-oriented programs (Zhao 2004).

Existing work on clustering software (Maletic and Marcus 2001; Kuhn et al. 2007), retrieving similar components in software libraries (Michail and Notkin 1999) and measuring semantic overlap of information in comments and identifiers among software modules (Etzkorn and Delugach 2000) uses the concept of semantic similarity between elements of the source code (Marcus et al. 2008), which stands at the foundation of the conceptual coupling, as defined in this paper.

2.2 The Use of IR Methods in Program Comprehension

IR methods were proposed and used successfully to address tasks of extracting and analyzing textual information existing in software artifacts. Early models were used to construct software libraries (Maarek et al. 1991; Fischer 1998) and support reuse tasks (Helm and Maarek 1991; Etzkorn and Davis 1997; Michail and Notkin 1999; Pan et al. 2004; Ye and Fischer 2005), while more recent work focused on specific software maintenance and development tasks such as recovery of traceability links. Several approaches have been proposed to recover traceability links between source code and external documentation using probabilistic IR, vector space models (Antoniol et al. 2002) and LSI (Marcus et al. 2005a). Other work proposed a set of approaches to recover traceability links among requirements (Clelang-Huang et al. 2005; Lo et al. 2006), requirements and source code (Hayes et al. 2006), requirements and test cases (Lormans and Van Deursen 2006), etc. A set of tools that integrates facilities to manage traceability links among different types of software artifacts was developed and evaluated recently (De Lucia et al. 2007).

IR methods have been also successfully used for concept and feature location (Marcus et al. 2004; Zhao et al. 2006; Poshyvanyk et al. 2007; Poshyvanyk and Marcus 2007; Eaddy et al. 2008) in the source code. Other approaches use IR methods to classify software systems based on their source code in open-source repositories (Kawaguchi et al. 2006) as well as cluster source code to obtain high-level views of software systems (Maletic and Marcus 2001; Kuhn et al. 2007).

IR techniques were also used to identify the starting impact set of a maintenance request (Antoniol et al. 2000), and to link change request descriptions to the set of historical file revisions impacted by similar past change requests (Canfora and Cerulo 2005). An approach to automatically classify the type of maintenance activity based on a textual description of changes was also proposed in (Mockus and Votta 2000). IR approaches have been used in the context of software measurement to assess the quality of identifiers and comments (Lawrie et al. 2006), measure complexity of the underlying software (Etzkorn et al. 2002), compute conceptual cohesion (Patel et al. 1992; Marcus and Poshyvanyk 2005) and coupling (Poshyvanyk and Marcus 2006) of classes.

In addition, IR techniques have been applied to several other tasks, such as identification of duplicate bug reports (Runeson et al. 2007; Wang et al. 2008), classification of software

maintenance requests (Di Lucca et al. 2002), recommendation rendering for novice programmers (Cubranic et al. 2005) and identification contributions of developers (Linstead et al. 2007).

2.3 Impact Analysis Approaches

During software change, programmers need to modify the source code of existing software systems. The first step during software change is to identify a part of the source code that needs to be changed. Once the starting point of the change is identified, developers need to identify the other components that need to be changed. Bohner et al. (Bohner 1996) recognized impact analysis as an activity that estimates all components to be changed. One of the techniques of impact analysis was proposed in the work of Queille et al. (Queille et al. 1994), where an interactive process was suggested, in which the programmer, guided by dependencies among program components (i.e., classes, functions), inspects components one-by-one and identifies the ones that are going to change—this process involves both searching and browsing activities. This interactive process was supported via a formal model, based on graph rewriting rules (Chen and Rajlich 2000).

More recent work appears in (Bohner and Gracanin 2003; Robillard 2005; Hill et al. 2007), where proposed tools can help navigate and prioritize system dependencies during various software maintenance tasks. The work in (Hill et al. 2007) relates to our approach in as much as it also uses lexical (textual) clues from the source code to identify related methods. Several recent papers presented algorithms that estimate the impact of a change on tests (Rountev et al. 2001; Kosara et al. 2003). A comparison of different impact analysis algorithms was provided in (Orso et al. 2004).

Coupling measures have been used to support impact analysis in OO systems (Briand et al. 1999a; Wilkie and Kitchenham 2000). Wilkie and Kitchenham (Wilkie and Kitchenham 2000) investigated if classes with high CBO coupling metric values are more likely to be affected by change ripple effects. Although CBO was found to be an indicator of change-proneness in general, it was not sufficient to account for all possible changes. The work of Briand et al. (Briand et al. 1999a) investigated the use of coupling measures and derived decision models for identifying classes likely to be changed during impact analysis. The results of empirical investigation of the structural coupling measures and their combinations showed that the coupling measures can be used to focus underlying dependency analysis and reduce impact analysis effort. On the other hand, the study revealed a substantial number of ripple effects, which are not accounted by the highly coupled (structurally) classes. This work motivated our quest for novel coupling measures, which use alternative sources of information (i.e., text in identifiers and comments) to capture dependencies that are not captured by the existing structural coupling measures.

3 Using IR Methods for Coupling Measurement

Our approach to coupling measurement is based on the hypothesis that modules (or classes) in (OO) software systems are related in more than one way. The evident 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. In this work, we propose a mechanism, based on IR techniques, to capture and measure this form of coupling, named as conceptual coupling. Our choice of IR technique in this type of application is LSI.

Developers use comments and identifiers to represent elements of the problem or solution domain of the software. In our previous work (Maletic and Marcus 2001; Marcus et al. 2004; Poshyvanyk and Marcus 2006; Poshyvanyk et al. 2007; Poshyvanyk and Marcus 2007; Marcus et al. 2008) we investigated approaches to extract, encode, and analyze the semantic information embedded in the comments and identifiers of the software. We use the same type of information in the definition of the conceptual coupling metrics.

In order to compute the conceptual coupling of classes, the source code of the software system is converted into a text corpus, where each document contains elements of the implementations of a method. Comments and identifiers are extracted from the source code, as well as structural information. The user has an option to choose the desired granularity (e.g., class or method level) for documents (see more details in Section 0). LSI uses this corpus to create a term-by-document matrix, which captures the distribution of words in methods. The main idea behind LSI is that the information about word contexts in which a particular word appears or does not appear provides a set of mutual constraints that determines the statistical similarity of meaning of sets of words to each other. LSI relies on a Singular Value Decomposition (SVD) (Salton and McGill 1983) of a term-by-document matrix derived from a corpus that pertains to knowledge in the particular domain of interest. SVD is applied to the term-by-document matrix to construct a subspace, called an LSI subspace. Each document from the corpus (i.e., method from the source code) is represented as a vector in this LSI subspace. Once the documents are represented in the LSI subspace, conceptual coupling measures can be computed between methods and classes. We use the cosine between the vectors corresponding to the methods as a measure of the conceptual coupling between the two methods.

The definition of and the methodology for measuring the conceptual coupling would not change radically if another IR method is to be used. The only significant change would be in the definition of the conceptual coupling between methods (see definition 3 in the next section).

3.1 System Representation and Coupling Measures

In order to define and compute the conceptual coupling measures, we introduce a graph based representation of a software system, similar to those used to compute other coupling measures.

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 $c \in C$, $M(c)=\{m_1, \ldots, m_z\}$ represents 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 (Conceptual Coupling Between Methods—CCM) The *conceptual coupling between two methods* $m_k \in M(C)$ and $m_j \in M(C)$, $CCM(m_k, m_j)$, 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.

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

As defined, the value of CCM(m_k, m_j) ∈ [−1, 1], as CCM is a cosine in the LSI space. In order to comply with non-negativity property of coupling metrics (Briand et al. 1999b), we refine CCM as:

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

Definition 4 (Conceptual Coupling Between a Method and a Class—CCMC) Let c_k ∈ C and c_j ∈ 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 conceptual coupling measure—CCM(m_k, m_j). We define the conceptual coupling between a method m_k and a class c_j as follows:

$$CCMC(m_k,\ c_j) = \frac{\sum_{q=1}^{t} CCM^1(m_k, m_{jq})}{t},$$

which is the average of the conceptual couplings between method m_k and all the methods from class c_j.

Definition 5 (Conceptual Coupling Between two Classes—CCBC) We define the conceptual coupling between two classes c_k ∈ C and c_j ∈ C as:

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

which is the average of the couplings between all unordered pairs of methods from class c_k and class c_j. The definition ensures that the conceptual coupling between two classes is symmetrical, as $CCBC(c_k,\ c_j)=CCBC(c_j,\ c_k)$.

3.2 The Conceptual Coupling of a Class

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

Definition 6 (Conceptual Coupling of a Class—CoCC) For a class c ∈ C, conceptual coupling is defined as:

$$CoCC(c) = \frac{\sum_{i=1}^{n} CCBC(c, d_i)}{n-1},$$

where $n=|C|$, d_i ∈ 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 coupling 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 (Briand et al. 1997).

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 from the source code of TortoiseCVS software system (see Fig. 1) with similarities between the methods outlined in Table 1. To simplify the example, we computed similarities only between a few methods in every class, however, in a real setting similarities will be computed for all pairs of methods among classes. We will refer to the class CVSServerFeatures as c_1 and to its methods as m_1 and m_2; to the class ConflictListDialog as c_2 and its methods as m_3, m_4, and m_5; to the class CommitDialog as c_3 and its methods as m_6, m_7, and m_8.

In order to compute CoCC for class c_1, we need to compute conceptual similarities between classes (c_1, c_2) and (c_1, c_3), since CoCC (c_1)=(CCBC(c_1, c_2)+CCBC(c_1, c_3))/2.

In order to compute the conceptual similarities between c_1 and c_2, we use the following formula: CCBC(c_1, c_2)=(CCMC (m_1, c_2)+CCMC (m_2, c_2))/2. In this case, CCMC(m_1, c_2) is an average of conceptual similarities between a method m_1 and all other methods in class c_2. Thus, CCMC(m_1, c_2)=(CCM^1(m_1, m_3)+CCM^1(m_1, m_4)+CCM^1(m_1, m_5))/3=(0.7+0.27+0.13) / 3=0.366. Similarly, CCMC (m_2, c_2)=(0.68+0.34+0.25)/3=0.423. Therefore, **CCBC(c_1, c_2)**=(0.366+0.423)/2=**0.3945**.

Similarly, we compute conceptual couplings between classes c_1 and c_3, **CCBC(c_1, c_3)**=**0.4515**.

Now we are able to compute CoCC(c_1), since **CoCC(c_1)**=(CCBC(c_1, c_2)+CCBC(c_1, c_3))/2=(0.3945+0.4515)/2=**0.423**. Similarly, **CoCC(c_2)=0.357** and **CoCC(c_3)=0.385.**

```
class CVSServerFeatures
{
public:
…
  CVSServerFeatures() {…};
  inline void SetCVSRoot(const std::string& cvsRoot){…};
…};

class ConflictListDialog : ResizeDialog
{
public:

   …
  ConflictListDialog(wxWindow* parent,
                const std::vector<std::string>& modified){…};
    void OnMenuMerge(wxCommandEvent& event){…};
    void OnMenuDiff(wxCommandEvent& e){…};
   …
 };
```

```
class CommitDialog : ResizeDialog
{
public:
   …
 CommitDialog(wxWindow* parent,
                const std::vector<std::string>& modified,
                const std::vector<std::string>& added,
                const std::vector<std::string>& removed,
                const std::string& defaultComment) { …};

   // Add files to ExtListCtrl

void AddFiles(const std::vector<std::string>& filenames,
                const std::vector<ItemData*>& itemData) {…};

static int wxCALLBACK CompareFunc(long item1,
                                  long item2,
                                  long sortData){ …}
 …};
```

Fig. 1 Source code of the CVSServerFeatures, CommitDialog, and ConflictDialog classes from the TortoiseCVS system

Table 1 Conceptual couplings between the methods of the classes CVSServerFeatures (m_1, m_2), ConflictListDialog (m_3, m_4, m_5), and CommitDialog (m_6, m_7, m_8). Conceptual couplings between methods of the same class

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
m_1	1	0.6	0.7	0.27	0.13	0.3	0.41	0.65
m_2	0.6	1	0.68	0.34	0.25	0.41	0.39	0.55
m_3	0.7	0.68	1	0.45	0.39	0.56	0.66	0.21
m_4	0.27	0.34	0.45	1	0.34	0.47	0.23	0.18
m_5	0.13	0.25	0.39	0.34	1	0.05	0.03	0.5
m_6	0.3	0.41	0.56	0.47	0.05	1	0.23	0.43
m_7	0.41	0.39	0.66	0.23	0.03	0.23	1	0.54
m_8	0.65	0.55	0.21	0.18	0.5	0.43	0.54	1

3.3 The Maximum Conceptual Coupling of a Class

If a class $c \in C$ has a high CoCC value, one can easily infer that it is strongly related to most other classes in the system. The opposite conclusion can be inferred if CoCC value is low. Little can be said if CoCC value is neither high nor low. It is a general drawback of average based metrics. In these cases we can still have classes strongly related to c, which are important from program comprehension point of view. These strong relationships can also propagate changes between classes.

An analogous logic can be applied to the coupling between two classes (e.g., if two methods from different classes are conceptually similar, they might need to be changed in concert).

With that in mind, we refine CoCC to capture only the strongest couplings among methods. The goal here is to make sure that our measuring mechanism does not miss classes that are highly coupled even to a part of the system, as developers need to be aware of such classes. Thus, we define:

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

The *maximum conceptual coupling between method* m_{kj} is denoted by the highest conceptual coupling among all possible pairs of methods between method m_k and all the methods in class c_j.

The maximum conceptual coupling between two classes based on $CCMC_m$ is defined as the following:

$$CCBC_m(\mathrm{c_k}, \mathrm{c_j}) = \frac{\sum_{l=1}^{r} CCMC_m(m_{kl}, c_j)}{r}$$

The *maximum conceptual coupling metric* $CoCC_m$ for a class c, is defined:

$$CoCC_m(c) = \frac{\sum_{i=1}^{n} CCBC_m(c, d_i)}{n-1},$$

where $n=|C|$, $d_i \in C$,$c \neq d_i$.

Referring back to the example in the previous subsection, with these new definitions, $\mathbf{CoCC_m(c_1)}$=(CCBC$_m$(c_1, c_2)+CCBC$_m$(c_1, c_3))/2=**0.645**. Similarly, $\mathbf{CoCC_m(c_2)}$=**0.486** and $\mathbf{CoCC_m(c_3)}$=**0.515.**

Class c_1 in our example is the one which has highest values of CoCC and CoCC$_m$ metrics, whereas class c_2 has the lowest conceptual coupling.

4 Comparing Structural and Conceptual Coupling Measures

As CoCC and CoCC$_m$ are new coupling measures, we evaluated them accordingly. In our previous work (Poshyvanyk and Marcus 2006) we analyzed theoretical properties of the proposed measures, such as, non-negativity, null value, monotonicity, merging of classes, and merging of unconnected classes. Additionally, we compared the conceptual coupling with existing structural coupling measures on ten different open source software systems. The key findings of those studies are presented in the following sub-sections.

4.1 Principal Component Analysis of Metrics Data

We compared the following set of coupling measures: nine structural (CBO, RFC, MPC, DAC, ICP, ACAIC, OCAIC, ACMIC, and OCMIC) and two conceptual coupling measures (CoCC and CoCC$_m$). In order to identify the causal, orthogonal dimensions captured by the coupling measures we performed Principal Component Analysis (PCA) (Jolliffe 1986) on the metrics measured on the set of 979 classes in ten open-source software systems. All studied measures were subjected to an orthogonal rotation. The results of the PCA revealed that the CoCC and CoCC$_m$ measures defined two new dimensions on their own (they were two separate significant factors in identified principal components). These results clearly indicated that conceptual 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, we compared the results of the PCA with those reported elsewhere in the literature (Briand et al. 1998; 2000). Although the PCs and loadings obtained in our case and those reported in the literature do not completely overlap, they were relatively similar (Poshyvanyk and Marcus 2006).

4.2 Differences Between Conceptual and Structural Coupling Measures

To obtain more insights into how the conceptual coupling metrics differ from the structural ones, we chose several classes from different software 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. In this subsection, we present some of the noted differences between the conceptual coupling measures (CoCC and CoCC$_m$) and the CBO and RFC structural coupling measures.

Table 2 Classes with highest conceptual coupling in WinMerge and TortoiseCVS according to CoCC and $CoCC_m$

System	Class	CoCC	CoCCm	CBO	RFC
WinMerge	IVSSItems	0.215	0.326	0	5
WinMerge	IVSSUsers	0.215	0.326	0	5
WinMerge	IVSSCheckouts	0.215	0.326	0	5
TortoiseCVS	ConflictListDialog	0.106	0.176	1	5
TortoiseCVS	ConflictParser	0.07	0.135	0	1

The classes chosen for detailed analysis are from the WinMerge[2] and TortoiseCVS[3] systems (Table 2). 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 the 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 a 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

[2] WinMerge is a visual text file differencing and merging tool for Windows and can be found at http://sourceforge.net/projects/winmerge (verified at 27/06/08)

[3] TortoiseCVS is a concurrent versions system (CVS) tool for Windows and can be found at http://sourceforge.net/projects/tortoisecvs(verified at 27/06/08)

exhibit a form of dependency, called hidden dependency (Yu and Vaclav 2001), 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 (Briand et al. 1999a).

4.3 Conceptual Coupling Between Pairs of Classes

From the impact analysis point of view, even more important are pairs of classes that relate conceptually, yet not structurally. To better understand the pair-wise conceptual coupling measures and how they can be used to rank classes during impact analysis, we also analyzed the CCBC and $CCBC_m$ measures, computed for pairs of classes in WinMerge and TortoiseCVS software systems. For illustration purposes, we selected several pairs of classes with highest CCBC and $CCBC_m$ values (see Table 3).

It came as no surprise that pairs of classes, mentioned in Section 4.2 as part of the "IVSS" cluster, were among those with highest CCBC values. These classes implement different, but related tasks, which are all based on implementation of client side of Object Linking and Embedding (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's services. 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 a history of common changes (i.e., they were changed and submitted to the repository at the same time).

Another pair of classes MergeDlg and UpdateDlg from TortoiseCVS system has high conceptual coupling values for CCBC and $CCBC_m$ 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 (Marcus and Maletic 2001).

5 Using Coupling Measures for Impact Analysis

The coupling measures can help order (rank) classes in software systems, based on different types of dependencies among classes, captured by the coupling measures (Briand et al. 1999a). Such coupling measures and derived ranks of classes can be computed automatically. The next section describes probabilistic decision models based on coupling measurement to support impact analysis.

Table 3 Pairs of classes from WinMerge and TortoiseCVS with highest CCBC values

System	Class	Class	CCBC	CCBCm
WinMerge	IVSSVersion	IVSSCheckout	0.776	0.964
WinMerge	IVSSItems	IVSSUsers	0.770	0.974
WinMerge	IVSSDatabase	IVSSCheckout	0.585	0.954
TortoiseCVS	MergeDlg	UpdateDlg	0.375	0.891

Table 4 Examples of redefined structural coupling measures used to rank classes during impact analysis

Name of the measure	Definition
CBO (coupling between object classes)	Two classes $c_i \in C$ and $d_i \in C$ are coupled to one another, if methods of one class use methods or attributes of the other, or vice versa. CBO is computed as a binary indicator, yielding 1 if c_i and d_i are coupled, else 0.
ICP (information-flow based coupling)	The number of method invocations in a class $c_i \in C$, of methods in a class $d_i \in C$, weighted by the number of parameters of the invoked methods. The measure also takes polymorphism into account.
DAC (data abstraction coupling)	The number of attributes in a class $c_i \in C$ that has class $d_i \in C$ as their type.

5.1 Ranking Classes Using Coupling Measures

For a given class $c \in C$ (which may be the starting point of a change, identified by the programmer based on his experience, or automatically with some feature location technique), the other classes in a software system are ranked according to their strength of coupling to the class c, based on a coupling measure or a combination of such measures (Briand et al. 1999a). The list of ranked classes is provided to the developer for further inspection. Since software systems may be large, sometimes containing thousands of classes, focusing impact analysis on strongly coupled classes may significantly reduce the burden on the developer.

In Section 2.1 we summarized the best known structural coupling measures. In the literature, these coupling measures are defined and used at the system level (classic definitions of coupling measures), meaning that they count, for a given class c, all dependencies (connections) from c to ***all*** other classes in the system. In order to use the coupling measures for impact analysis, they need to be modified to account for coupling between ***pairs*** of classes only. Table 4 shows how we redefined some of the structural coupling measures. More details on how other structural coupling measures are redefined on a class pair-wise basis are provided by Briand et al. (Briand et al. 1999a). Section 3.1 provides details on how we defined conceptual coupling measures on pair-wise basis.

5.2 An Example of Using Coupling Measures for Impact Analysis in Mozilla

The following example illustrates how conceptual and structural coupling metrics are used to rank classes to focus impact analysis. The bug #232570[4] reports some problems associated with 'ldap2.server.position values for ab pane and search order' in Mozilla. In order to fix the bug, the developer needs to find and change the classes in the source code containing this bug. Assume that the starting point of this change, the class *nsAbDirectoryQuery*, is identified via some available feature location technique. Given the starting point, the developer needs to perform impact analysis to identify the remaining classes in order to complete the change. In our approach, we compute the set of pair-wise coupling measures for all possible pairs between *nsAbDirectoryQuery* and other classes. Using these coupling measures, all the classes in Mozilla are ranked based on the strength of coupling

[4] The bug can be accessed in Bugzilla at https://bugzilla.mozilla.org/show_bug.cgi?id=232570 (verified at 27/06/08)

(different type of couplings are captured by different measures) to the *nsAbDirectoryQuery* class. The idea is that the strongly coupled classes to the given class are more likely to change (Briand et al. 1999a). In our example, Table 5 provides the list of top classes ranked by the values of two coupling metrics, $CCBC_m$ and ICP. These measures provide the quantitative estimation of the strength of coupling between the class *nsAbDirectoryQuery* and the classes in Table 5. In order to determine the number of candidate classes suggested for inspection during impact analysis, different strategies can be used. The most common approaches are to use a cut point *cp* (i.e., select the top *n* classes from the list or the top *n%*) or a threshold *t* (i.e., select all classes that have a coupling value higher/lower than some metric value *t*). Combinations of the two approaches are also used. For example the top *n%* classes will be retrieved if they have a coupling value higher or lower than *t*.

In this example, for each metric, a cut point strategy is used (e.g., the top five classes from each rank list are retrieved).

While using $CCBC_m$ for ranking conceptually similar classes to *nsAbDirectoryQuery* class, we retrieve five out of 4,853 (see Table 5). Two of these classes, *nsAbMDBDirectory* and *nsAbLDAPDirectory*, are among those ten classes in the official patch that were changed to fix this bug (*nsAbAutoCompleteSession, nsAbBSDirectory, nsAbCardProperty, nsAbDirProperty, nsAbDirectoryDataSource, nsAbDirectoryProperties, nsAbDirectory-Query, nsAbLDAPDirectory, nsAbMDBDirectory, nsMsgCompose*). However, when ICP metric is used with this cut point, only two classes are suggested and none of them is among the changed classes. The precision and recall for these two metrics is computed as the following. Precision for $CCBC_m$ is 2/5*100%=40%, while recall is 2/9*100=22% (we use nine classes instead of ten in the denominator, since one of the changed classes, *nsAbDirectoryQuery*, is already identified and used as a starting point in impact analysis). None of the changed classes has structural dependencies, which are captured by ICP coupling measure, with *nsAbDirectoryQuery* class and thus precision and recall for ICP measure is zero.

6 Case Study on Using Coupling Measures to Support Impact Analysis

In this section we present a case study, where we empirically investigated how conceptual coupling metrics can be used during impact analysis as well as compared them to a set of existing structural coupling measures used for the same task.

Table 5 Classes strongly coupled with *nsAbDirectoryQuery* and ranked according to $CCBC_m$ and ICP coupling measures

Rank	$CCBC_m$		ICP	
	Classes	Values	Classes	Values
1	nsAbQueryLDAPMessageListener	0.86	nsDebug	123
2	***nsAbMDBDirectory***	0.81	nsAFlatString	121
3	nsAbDirectoryQuerySimpleBoolExpression	0.79		
4	***nsAbLDAPDirectory***	0.76		
5	nsAbView	0.72		

The top five classes are listed for **$CCBC_m$**. Only two top classes are listed for ICP measure, since only these two classes are structurally connected to *nsAbDirectoryQuery* according to ICP measure. The classes in bold are those that actually changed together with class nsAbDirectoryQuery to fix the bug# 232570

6.1 Design of the Case Study

The case study is designed in a similar fashion to the one presented in the work by Briand et al. (Briand et al. 1999a), where a set of structural coupling metrics was used to rank classes during impact analysis in an OO system. While designing and conducting the case study, we followed the guidelines in two papers written by Yin and by Flyvbjerg, respectively (Yin 2003; Flyvbjerg 2006).

6.1.1 Objectives and Methodology

In this case study, the CCBC and $CCBC_m$ measures are compared with nine existing structural coupling measures (i.e., PIM, ICP, CBO, MPC, OCMIC, DAC, OCAIC, ACMIC, and ACAIC) to evaluate whether they provide better support for impact analysis or not. The premise is that given the nature of the captured information (e.g., textual information in identifiers and comments) and the counting mechanism employed by CoCC and $CoCC_m$, these measures should capture different aspects of coupling among classes as compared to the nine existing coupling metrics, which utilize only structural information.

In the case study, we used the source code of Mozilla v1.6, which is an open-source web browser ported on almost all known software and hardware platforms. It is large enough to represent a real-world software system and it comes with an available history of changes. The source code of Mozilla consists of 4,853 classes implemented in approximately four million lines of source code (including 738,180 lines of comments).

Our case study addresses the following question: *Do CCBC and $CCBC_m$ provide better support for ranking classes during impact analysis than any of the following structural coupling measures: PIM, ICP, CBO, MPC, OCMIC, DAC, OCAIC, ACMIC and ACAIC?*

6.1.2 Settings of the Case Study

All the structural coupling measures, including pair-wise versions of coupling measures, were computed using *Columbus* (Ferenc et al. 2002). The conceptual coupling measures were computed with the *IRC^2M* tool (Poshyvanyk and Marcus 2006), which can be used with several settings for the underlying LSI-based analysis. In the case studies, we used the following common settings.

We used a method level granularity, to construct the corpus for Mozilla, meaning that the implementation (source code) of every method from the software system was extracted and represented as a separate document in the corpus. We extracted all types of methods from classes in the source code, including constructors, destructors, and accessors. Comments and identifiers were extracted from each method as well. The resulting text from source code is pre-processed as the following: some of the tokens are eliminated (e.g., operators, special symbols, some numbers, keywords of the C++ programming language, standard library function names including standard template library, etc.); the identifier names in the source code are split into parts based on observed coding standards and naming conventions. For example, all the following identifiers are broken into separate words 'coupling' and 'measures': 'coupling_measures', 'Coupling_measures', 'CouplingMeasures', etc. Given that we did not consider n-grams, the order of words in text passages is of no significance. In this process, LSI does not use a predefined vocabulary, or a predefined grammar, therefore no morphological analysis or transformations are required, such as stemming or abbreviation expansion. We believe that such methods may even improve the results. Based on our previous experience with LSI on corpus of similar size (Marcus and

Poshyvanyk 2005; Poshyvanyk et al. 2007; Marcus et al. 2008), we used a reduction factor of 500 for the Mozilla software system corpus.

6.1.3 Collecting Change Data in Mozilla for Evaluation

In order to compare conceptual and structural coupling measures for identifying classes that change together (i.e., changes related to the same bug report and having the same identification number in the configuration management system) during impact analysis, we utilized the history of changes in Mozilla. We used Bugzilla[5], a bug-tracking system used in the development of Mozilla and collected the bugs between two versions of Mozilla (i.e., 1.6 and 1.7) and correlated each bug with specific classes. The Bugzilla database contained around 256,613 different bug entries for all the versions of the system, however, we restricted the scope of mining only to the bugs which appear between versions 1.6 and 1.7 and were fixed, meaning that the bug was officially closed and contained an official patch file with modifications). In our analysis we did not consider bug reports for accessory software systems such as Bonsai, Tinderbox, etc. We extracted 1,021 different bug entries which satisfied all the aforementioned requirements. By analyzing the patch files, associated with bug reports, we assigned bugs to particular intervals in the source code. This was possible to complete automatically, since each patch file contained the name of the changed file and it described how many lines were deleted from a given line number and how many lines were inserted at a given line number. Using this line-level information about changes, we determined intervals of actual changes in the file and localized the bugs to implementations in the source code of specific classes. To ensure that the files in the patch were changed at the same time, we searched and checked log messages in the configuration management system to ensure that check-in messages for those files contained the same identification number, assigned by the Bugzilla system.

After collecting a set of bug reports and sets of changed classes respectively, we filtered the data to eliminate those bugs which contained only one modified class. After the filtering we ended up with 391 bug reports, containing on average 7.3 modified classes (standard deviation: 14.6). However, we also removed some outliers in the data. For example, one of the removed outliers, the patch in the bug report #226439[6], contained the record number of modified classes (149) to fix this bug.

6.1.4 Evaluation Methodology

Our evaluation strategy is to utilize the history of changes, observed in Mozilla, to determine whether existing structural and conceptual coupling measures can be used during impact analysis to identify classes with common changes (i.e., changes in classes related to the same bug report and having the same bug identification number in the configuration management system). The history of changes can be used to evaluate rankings of classes produced with different coupling measures against actual changes in the software system. We *expect* that the conceptual coupling measures, namely CCBC and $CCBC_m$, will be at

[5] Bugzilla is a web-based general-purpose bug-tracking tool originally developed and used by the Mozilla project, and licensed under the Mozilla Public License. Bugzilla can be fount at http://bugzilla.mozilla.org/ (verified at 27/06/08)

[6] The bug can be accessed in Bugzilla at https://bugzilla.mozilla.org/show_bug.cgi?id=226439 (verified at 27/06/08)

least as effective as the nine existing coupling measures in ranking classes during impact analysis.

The evaluation methodology can be summarized in the following steps:

- For a given software system, a set of bug reports $B=\{b_1, b_2 \ldots b_n\}$ is mined from the bug tracking system. The set of classes, which had been changed to fix each bug (e.g., $c(b_1)=\{c_1, c_2 \ldots c_n\}$) are mined from the configuration management system. Specific details on how the bug reports and changed classes are identified are provided in Section 6.1.3.
- For each class in $c(b_i)$, pair-wise structural and conceptual coupling metrics are computed. The values of each metric are used to compute ranks of the remaining classes in the software system.
- Using a specific cut point criteria (which ranges anywhere from 10 to 500 classes), defined as *cp*, select top *n* classes in each ranked list of results generated by every metric. For every class in $c(b_i)$, which is used in the evaluation, we assess the effectiveness of identifying relevant classes (i.e., the other classes in $c(b_i)$) via rankings of specific coupling metric.
- In order to evaluate each coupling measure and compare all the coupling measures used in the case study, the suggested ranked lists of classes are compared against classes that were actually changed. Average precision (*P*), recall (*R*), and *F*-measure (*F*) for each class in $c(b_i)$ for each $i=1..|B|$ are computed for every metric. In our case, *precision* is the percentage of classes suggested by a metric that are actually changed together with the given class according to the bug report. *Recall* is the percentage of the classes that are changed together with the given class and are successfully retrieved using the coupling measures (see Section 5.2 for an example on how precision and recall values are computed). The *F-measure* is a weighted harmonic mean of precision and recall and calculated as (2 × precision × recall) / (precision+recall) and can be used as a comprehensive indicator of combined precision and recall values. The *F*-measure is better suited than techniques like averaging, since it weights the lower measure more heavily. For example, a coupling measure producing 80% precision but only 20% recall provides only a few suggested classes, but most of these classes are relevant. The *F*-measure for this case is 32%, whereas the average of precision and recall is 50%. Thus *F*-measure (32%) better reflects a coupling measure's effectiveness, since the measure helps to identify only 20% of the relevant classes. For each measure, a higher value is more desirable.

The complete results on using conceptual and structural coupling measures to rank classes for all mined bug reports are presented and discussed in Section 6.2.

6.2 Comparing Conceptual and Structural Coupling Metrics for Impact Analysis

In order to compare the coupling measures, we followed the evaluation methodology presented in Section 6.1.4. We computed precision and recall values for each coupling metric for every class in each of the 391 bug reports. We computed 1,490 precision and recall values for eleven coupling measures. In addition, we studied the impact of different cut points, on precision, recall and *F*-measure values for particular coupling measures. We computed precision, recall and *F*-measure values for each cut point for each coupling metric.

The results of using nine structural and two conceptual coupling measures to rank classes during impact analysis in Mozilla are presented in Tables 6, 7, 8. The values of the

Table 6 Precision (Pre) and recall (Rec) values for using conceptual and structural coupling measures to rank classes during impact analysis based on different cut points from 10 to 50 classes

	Cut point=10			Cut point=20			Cut point=30			Cut point=40			Cut point=50		
	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr
CCBCm	27.8	14.6	0.64	24.7	22.1	0.61	22.4	27.7	0.60	20.3	31.7	0.59	18.36	34.5	0.59
ICP	11.9	6.9	268	10.1	9.7	268	9.3	12.0	268	8.8	14.2	268	8.6	16.5	268
PIM	11.3	6.60	138	9.84	9.56	138	9.13	11.7	138	8.66	13.8	138	8.52	16.3	138
CCBC	10.8	5.6	0.47	9.5	8.9	0.45	8.1	11.1	0.45	7.2	12.8	0.44	6.7	14.1	0.44
CBO	7.2	6.2	10	5.4	9.4	20	4.1	10.4	30	3.3	10.9	40	2.8	11.3	50
MPC	6.6	5.7	3	3.9	6.7	0	2.8	7.0	0	2.1	7.0	0	1.7	7.0	0
OCMIC	2.0	2.1	2	1.1	2.2	0	0.8	2.3	0	0.6	2.3	0	0.5	2.3	0
OCAIC	1.7	2.0	0	1.0	2.1	0	0.6	2.1	0	0.5	2.1	0	0.4	2.1	0
DAC	1.8	2.0	0	1.0	2.1	0	0.6	2.1	0	0.5	2.1	0	0.4	2.1	0
ACMIC	0.9	0.4	0	0.5	0.4	0	0.3	0.4	0	0.2	0.4	0	0.2	0.4	0
ACAIC	0.8	0.3	0	0.4	0.3	0	0.3	0.3	0	0.2	0.3	0	0.2	0.3	0

The values of pair-wise conceptual and structural coupling measures taken at each cut point are provided in Threshold (Thr) column

computed coupling metrics for classes in Mozilla can be downloaded from http://www.cs.wayne.edu/~severe/CoCC/Mozilla_coupling_metrics.zip. The results in Tables 6, 7, 8 contain precision and recall values for using $CCBC_m$, ICP, PIM, CCBC, CBO, MPC, OCMIC, OCAIC, DAC, ACMIC and ACAIC coupling measures with different cut points ranging from 10 classes to 500 classes.

Only two of the coupling metrics, CCBC and $CCBC_m$, are normalized (see Section 3.2), thus we could compute precision, recall, and *F*-measure values for various thresholds within the complete specter of metric values (see Fig. 2). The other coupling metrics are not normalized as they count the total number of coupling connections of a class with other classes in the system (the larger the metric value, the stronger the coupling between two classes). The only exception is CBO coupling measure, which has a binary value indicating if two classes have a coupling connection or not. In case of CBO, we based our evaluation

Table 7 Precision (Pre) and recall (Rec) values for using conceptual and structural coupling measures to rank classes during impact analysis based on different cut points from 60 to 100 classes

	Cut point=60			Cut point=70			Cut point=80			Cut point=90			Cut point=100		
	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr
CCBCm	16.6	36.5	0.58	15.3	38.6	0.58	14.2	40.2	0.57	13.4	41.8	0.57	12.62	43.1	0.57
ICP	8.5	18.8	268	7.9	20.1	157	7.3	20.9	69	6.8	21.8	29	6.5	22.8	22
PIM	8.43	18.7	138	7.82	19.9	69	7.23	20.6	30	6.81	21.6	13	6.52	22.6	11
CCBC	6.3	15.5	0.43	5.9	16.5	0.43	5.6	17.7	0.43	5.4	18.8	0.43	5.2	19.8	0.42
CBO	2.4	11.6	60	2.1	11.7	70	1.9	11.8	80	1.7	11.9	90	1.6	12.0	100
MPC	1.4	7.1	0	1.3	7.2	0	1.1	7.2	0	1.0	7.2	0	0.9	7.2	0
OCMIC	0.4	2.4	0	0.4	2.5	0	0.3	2.5	0	0.3	2.5	0	0.3	2.5	0
OCAIC	0.3	2.2	0	0.3	2.3	0	0.3	2.3	0	0.3	2.3	0	0.2	2.3	0
DAC	0.3	2.2	0	0.3	2.3	0	0.3	2.3	0	0.3	2.3	0	0.2	2.3	0
ACMIC	0.2	0.5	0	0.2	0.6	0	0.2	0.6	0	0.2	0.6	0	0.1	0.6	0
ACAIC	0.1	0.4	0	0.2	0.5	0	0.2	0.5	0	0.1	0.5	0	0.1	0.5	0

The values of pair-wise conceptual and structural coupling measures taken at each cut point are provided in Threshold (Thr) column

Table 8 Precision (Pre) and recall (Rec) values for using conceptual and structural coupling measures to rank classes during impact analysis based on different cut points from 200 to 500 classes

	Cut point=200			Cut point=300			Cut point=400			Cut point=500		
	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr	Pre	Rec	Thr
CCBCm	8.39	52.4	0.55	6.47	58.2	0.54	5.35	62.1	0.53	4.61	65.1	0.52
ICP	4.13	27.3	22	3.26	31.1	22	2.89	34.9	22	2.63	39.0	22
PIM	4.12	27.1	11	3.25	30.8	11	2.88	34.8	11	2.62	38.9	11
CCBC	3.99	27.0	0.41	3.44	33.3	0.39	3.16	39.1	0.38	2.98	44.8	0.36
CBO	1.05	13.2	200	1.01	16.1	300	1.00	20.4	400	0.99	26.5	500
MPC	0.72	8.56	0	0.81	11.7	0	1.07	16.3	0	1.22	22.7	0
OCMIC	0.47	4.25	0	0.67	7.98	0	0.97	12.8	0	1.19	20.2	0
OCAIC	0.45	4.15	0	0.66	7.89	0	0.96	12.5	0	1.18	19.9	0
DAC	0.19	2.4	0	0.18	2.41	0	0.18	2.41	0	0.17	2.42	0
ACMIC	0.40	2.41	0	0.64	6.36	0	0.94	11.1	0	1.17	18.5	0
ACAIC	0.40	2.38	0	0.64	6.32	0	0.94	11.1	0	1.17	18.5	0

The values of pair-wise conceptual and structural coupling measures taken at each cut point are provided in Threshold (Thr) column

on choosing *n* coupled classes to a given class instead of using actual metric values (as it is done in cases of other structural coupling measures).

In case of each coupling measure we varied a cut point from 10 to 500 classes. For instance, in case of using $CCBC_m$ metric (see Table 6), with a cut point of 10 classes, obtained precision was 27.80%, recall was 14.6% and *F*-measure was 19.1%. Increasing a cut point to 20 classes provides more candidate classes, thus decreasing precision to 24.7%, but significantly increasing recall values to 22.1% and increasing *F*-measure to 23.3%. Also notice that while using a cut point of 10 classes, the $CCBC_m$ value for a class taken at a cut point is 0.64, however while using a cut point of 20 classes the class at a cut point has a smaller $CCBC_m$ value of 0.61, meaning that conceptual similarities for ten of the candidate classes are within [0.61 and 0.64] interval of $CCBC_m$ metric values.

Analysis of the results, presented in Tables 6, 7, 8, 9 reveals that $CCBC_m$ conceptual coupling metric is the best coupling measure for ranking classes during impact analysis (in terms of precision, recall and *F*-measure). None of the other coupling metrics achieves the same value of *F*-measure (i.e., maximum of 24.8% for a cut point of 30/40 classes and 19.0% on average across all the cut points) for any given cut point. For example, when using a cut point of 30 classes, using $CCBC_m$, around 28% of actually changed classes are recovered (*recall*) and one in five suggested classes is correct (*precision*). These are encouraging results as the source code of Mozilla consists of 4,853 classes and focusing developers on a set of relevant classes can significantly reduce amount of time developers spend on impact analysis.

The results for using structural coupling measures for the same task are less encouraging. The second best metric after $CCBC_m$ is the structural coupling measure ICP (based on the average *F*-measure, see Table 9), which captures information flow based coupling. This coupling measure captures the number of invocations in a class $c_i \in C$, of methods in a class $d_i \in C$, weighted by the number of parameters of the invoked methods. This coupling measure also takes polymorphism into account. While using the cut point of 20 classes, the precision of identifying relevant classes using ICP is 10.1%, recall is 9.7% and *F*-measure is 9.89%. The best value of *F*-measure for ICP metric, which is 11.7%, is obtained while using a cut point of 60 classes (see Table 9).

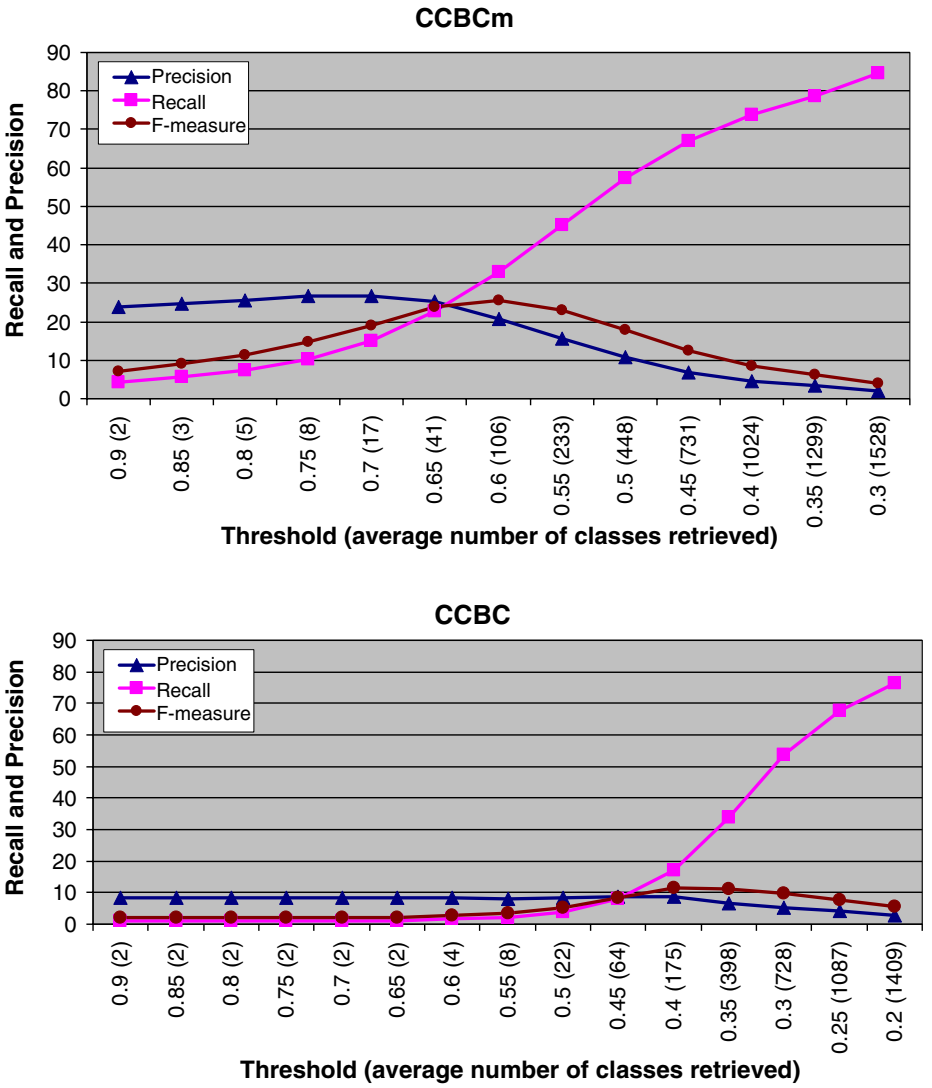

Fig. 2 Precision, recall and *F*-measure values for using CCBCm and CCBC conceptual coupling measures to rank classes during impact analysis. The values are based on different thresholds (pertinent to each metric). The number of actually retrieved classes for every threshold is given in *parenthesis*

The next metric after ICP is PIM (see Table 9), which captures the number of method invocations in class $c_i \in C$ of methods in class $d_i \in C$. The measure also takes polymorphism into account. For example, when using the first twenty classes with the highest PIM values as a cut point, the precision of identifying relevant classes is 9.84%, achieved recall is 9.56 while *F*-measure is only 9.7%. The best value of *F*-measure for PIM metric, which is 11.6%, is obtained while using a cut point of 60 classes (see Table 9). PIM has been shown to be a relatively effective coupling measure (as compared to other structural measures) to rank classes during impact analysis in other case studies (Briand et al. 1999a).

Table 9 *F*-measure values for using conceptual and structural coupling measures to rank classes during impact analysis based on different cut points from 10 to 500 classes

	10	20	30	40	50	60	70	80	90	100	200	300	400	500	Avg
CCBCm	19.1	23.3	*24.8*	*24.8*	23.9	22.9	21.9	21.1	20.3	19.5	14.5	11.6	9.86	8.62	19.0
ICP	8.77	9.89	10.5	10.8	11.3	*11.7*	11.3	10.8	10.4	10.1	7.19	5.91	5.34	4.93	9.22
PIM	8.35	9.70	10.3	10.7	11.2	*11.6*	11.2	10.7	10.4	10.1	7.15	5.89	5.33	4.93	9.12
CCBC	7.34	9.18	*9.38*	9.24	9.05	8.94	8.71	8.51	8.39	8.27	6.95	6.24	5.85	5.59	7.97
CBO	6.70	*6.85*	5.92	5.05	4.45	3.95	3.61	3.28	3.01	2.77	1.96	1.90	2.28	2.54	3.88
MPC	*6.16*	4.97	3.99	3.24	2.73	2.36	2.15	1.93	1.74	1.59	1.34	1.52	2.01	2.33	2.72
OCMIC	2.06	1.48	1.15	0.92	0.77	0.67	0.68	0.61	0.55	0.50	0.85	1.23	1.80	*2.25*	1.11
OCAIC	1.85	1.32	0.98	0.78	0.65	0.56	0.59	0.53	0.48	0.43	0.82	1.23	1.79	*2.24*	1.02
DAC	*1.88*	1.34	0.93	0.81	0.67	0.53	0.53	0.53	0.53	0.37	0.35	0.33	0.33	0.32	0.68
ACMIC	0.56	0.43	0.35	0.29	0.25	0.23	0.29	0.26	0.24	0.22	0.69	1.16	1.73	*2.20*	0.64
ACAIC	0.43	0.33	0.27	0.23	0.20	0.19	0.26	0.23	0.21	0.20	0.69	1.15	1.73	*2.20*	0.60

The coupling measures are ranked in descending order based on the average value of *F*-measures computed at each cut point. The largest *F*-measures for each metric are italicized

The MPC coupling measure shows higher precision values as compared to other coupling measures in some cases (more than 7%), however it has low recall (around 2% on average) for all of the studied cut points.

The other coupling measures, such as CBO, DAC, ACAIC, ACMIC, OCAIC and OCMIC have low precision and recall values (less than 10%) for all of the computed cut points.

While CCBC coupling measure uses the same type of information as $CCBC_m$, it uses a different counting mechanism based on average similarities as opposed to $CCBC_m$, which is based on the strongest coupling link between classes. According to the results (see Tables 6, 7, 8, 9), $CCBC_m$ significantly outperforms CCBC. Moreover CCBC is outperformed by some of the structural coupling measures such as ICP and PIM.

The results show that $CCBC_m$ is a useful (the best among the studied coupling measures) indicator of an external property of classes in OO systems-change proneness. This coupling measure can be effectively used to rank relevant classes during impact analysis in OO systems. The measure performed better on average than any of the structural metrics we compared it to. While we do not investigate to which extent structural and conceptual coupling measures complement each other in this case study, there is a noticeable potential in combining these coupling measures for ranking classes during impact analysis. As it has been observed in several examples in Section 4.2 there are cases where high conceptual coupling metric values capture dependencies between classes, which are not structurally connected and vice versa. Thus, combining conceptual and structural coupling measures may lead to significant reduction of programmer's efforts during impact analysis via increasing precision (and recall) of identifying relevant classes.

6.3 Testing Statistical Significance of Differences Among Precision and Recall Values

In order to compare values of precision and recall for the coupling measures for each of the cut points and conclude whether or not the difference is statistically significant, we executed the Kruskal–Wallis's test, which is a nonparametric alternative to the one-way analysis of variance (ANOVA) in those cases when more than three independent samples are present. In our case, the Kruskal–Wallis's test is an appropriate alternative to ANOVA

test, because we have eleven independent samples of precision and recall values for each of the coupling measures.

We executed Kruskal–Wallis's test separately for precision and recall values for all of the coupling measures (see Table 10). For more details on the Kruskal–Wallis's test, the reader is referred to the work of Siegel and Castellan (Siegel and Castellan 1988).

In the both tests, at the level of significance for alpha=0.05, the decision was to reject the null hypothesis of absence of differences between even metric values. In other words, both tests have shown that the differences between precision (first test) and recall (second test) values for eleven coupling metrics were statistically significant.

6.4 Threats to Validity

We identify several issues that affected the results of our case study and limited our interpretations.

In the case study we considered only structural metrics that were based on the static information obtained from the source code. The results can differ to some extent if dynamic coupling measures are used (Arisholm et al. 2004; Mitchell and Power 2006).

The conceptual coupling measures depend on rational naming conventions for identifiers and comments in source code. When these are missing, the only hope for measuring any aspects of coupling rests on the structural coupling measures.

CoCC, $CoCC_m$, CCBC and $CCBC_m$ measures, as currently defined, do not take into account polymorphism and inheritance. The measures only consider methods of a class that are implemented or overloaded in the class. One of the solutions, which accounts for inheritance, consists of extending the measures to include the source code of inherited methods into the documents of derived classes, as it is currently done by Kuhn et al. (Kuhn et al. 2007).

In our case study we used one large software system, however, to allow for generalization of results, large-scale evaluation is necessary, which should take into account several releases of software systems from different domains, developed using different programming languages.

Also, our evaluation is based on the changed classes extracted from patches in related bug reports. This could have impacted evaluation procedure as these patches may contain incomplete information about actually changed classes or these changes could have introduced other bugs. We alleviate this issue by considering only patches which are officially approved by module owners in Mozilla.

Table 10 The results of running two Kruskal–Wallis tests for precision (test 1) and recall (test 2) values of eleven coupling metrics across the different cut points

	Test 1 precision	Test 2 recall
H (observed value)	126.55	110.905
H (critical value)	18.31	18.307
DF (degrees of freedom)	10	10
One-tailed *p*-value	<0.0001	<0.0001
Alpha	0.05	0.05

7 Conclusions and Future Work

The paper defines a novel set of operational measures for the conceptual coupling of classes, based on IR, which are theoretically valid and empirically studied. These new metrics capture new dimensions in coupling measurement, compared to existing structural metrics. Moreover, one of the conceptual coupling measures, $CCBC_m$ measure, appears to be a superior indicator of change ripple effects as compared to existing structural coupling measures and can be effectively used to rank classes in the course of impact analysis in a large OO system.

The paper lays the foundation for a wealth of work that makes use of the coupling metrics which use lexical (textual) information in software. The proposed metrics could be further extended and refined, for example by taking into account inheritance in measurement. The IRC^2M tool will be adapted to compute conceptual coupling measures in other programming languages such as Java or C#. We are also planning on comparing and combining the conceptual coupling metrics with the evolutionary based coupling (Gall et al. 2003). Since conceptual coupling measures use textual information, we are considering including external documentation in the corpus. This will allow extending the context in which words are used in the software and identifying inconsistencies between source code and external documentation.

More importantly, we will investigate combinations of the conceptual and structural coupling measures for impact analysis and detection of hidden dependencies. In addition, we will use these metrics to extend prior work on software clustering (Kuhn et al. 2007), concept location (Marcus et al. 2004; Marcus et al. 2005b; Poshyvanyk and Marcus 2007), and high-level concept clone detection (Marcus and Maletic 2001). We are also planning on investigating how changes in the structure and lexicon of software during software evolution (Antoniol et al. 2007) are reflected in structural and conceptual coupling measures.

Acknowledgements This research was supported in part by grants from the U.S. National Science Foundation (CCF-0438970 and CCF-0820133), by the Hungarian national grants GVOP-3.3.1.-2004-04-0024/3.0 and GVOP-3.1.1.-2004-05-0345/3.0 and by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. We would like to thank the anonymous reviewers for their pertinent and helpful comments.

References

Abreu F, Pereira G, Sousa P (2000) A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. Conference on Software Maintenance and Reengineering (CSMR'00). IEEE Computer Society, Zurich Switzerland, pp 13–22

Allen EB, Khoshgoftaar TM, Chen Y (2001) Measuring coupling and cohesion of software modules: an information-theory approach. 7th International Software Metrics Symposium (METRICS'01), 124–134.

Antoniol G, Fiutem R, Cristoforetti L (1998) Using Metrics to Identify Design Patterns in Object-Oriented Software. 5th IEEE International Symposium on Software Metrics (METRICS'98), Bethesda, MD, 23–34.

Antoniol G, Canfora G, Casazza G, Lucia A (2000) Identifying the Starting Impact Set of a Maintenance Request: A Case Study. 4th European Conference on Software Maintenance and Reengineering (CSMR2000), Zurich, Switzerland, 227–231.

Antoniol G, Canfora G, Casazza G, De Lucia A, Merlo E (2002) Recovering traceability links between code and documentation. IEEE Trans Softw Eng 28(10):970–983

Antoniol G, Gueheneuc Y-G, Merlo E, Tonella P (2007) Mining the Lexicon Used by Programmers during Software Evolution. 23rd IEEE International Conference on Software Maintenance (ICSM'07). IEEE Computer Society, Paris, France, pp 14–23

Arisholm E, Briand LC, Foyen A (2004) Dynamic coupling measurement for object-oriented software. IEEE Trans Softw Eng 30(8):491–506

Bohner S (1996) Impact analysis in the software change process: A year 2000 perspective. International Conference on Software Maintenance (ICSM '96), Monterey, CA, 42–51

Bohner SA, Gracanin D (2003) Software impact analysis in a virtual environment. Software Engineering Workshop, 143–151.

Briand L, Wüst J, Louinis H (1999a) Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. IEEE International Conference on Software Maintenance (ICSM'99), IEEE Computer Society Press, 475–482

Briand LC, Devanbu P, Melo WL (1997) An investigation into coupling measures for C++. International Conference on Software engineering (ICSE'97). ACM, Boston, pp 412–421

Briand LC, Daly JW, Porter V, Wüst J (1998) A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems. 5th International Software Metrics Symposium (METRICS'98), Bethesda, MD, IEEE Computer Science, 43–53

Briand LC, Daly J, Wüst J (1999b) A unified framework for coupling measurement in object oriented systems. IEEE Trans Softw Eng 25(1):91–121

Briand LC, Wüst J, Daly JW, Porter VD (2000) Exploring the relationship between design measures and software quality in object-oriented systems. J Syst Softw 51(3):245–273

Canfora G, Cerulo L (2005) Impact Analysis by Mining Software and Change Request Repositories. 11th IEEE International Symposium on Software Metrics (METRICS'05), 20–29.

Chen K, Rajlich V (2000) Case Study of Feature Location Using Dependence Graph. 8th IEEE International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, 241–249.

Chidamber SR, Kemerer CF (1991) Towards a Metrics Suite for Object Oriented Design. Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'91), SIGPLAN Notices, 197–211.

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20 (6):476–493

Clelang-Huang J, Settimi R, Duan C, Zou X (2005) Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability. International Requirements Engineering Conference (RE'05), Paris, France, 135–144.

Cubranic D, Murphy GC, Singer J, Booth KS (2005) Hipikat: a project memory for software development. IEEE Trans Softw Eng 31(6):446–465

De Lucia A, Fasano F, Oliveto R, Tortora G (2007) Recovering Traceability Links in Software Artefact Management Systems. ACM Transactions on Software Engineering and Methodology 16(4).

Deerwester S, Dumais ST, Furnas GW, Landauer TK, Harshman R (1990) Indexing by Latent Semantic Analysis. J Am Soc Inf Sci 41:391–407

Di Lucca GA, Di Penta M, Gradara S (2002) An Approach to Classify Software Maintenance Requests. IEEE International Conference on Software Maintenance (ICSM'02), Montréal, Québec, Canada, 93–102.

Eaddy M, Aho AV, Antoniol G, Guéhéneuc YG (2008) CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis. 17th IEEE International Conference on Program Comprehension (ICPC'08), Amsterdam, The Netherlands.

El-Emam K, Melo K (1999) The Prediction of Faulty Classes Using Object-Oriented Design Metrics. NRC/ERB-1064 NRC 43609.

Etzkorn L, Delugach H (2000) Towards a semantic metrics suite for object-oriented design. 34th International Conference on Technology of Object-Oriented Languages and Systems, 71–80.

Etzkorn LH, Davis CG (1997) Automatically identifying reusable OO legacy code. IEEE Computer 30 (10):66–72

Etzkorn LH, Gholston S, Hughes WE (2002) A semantic entropy metric. Journal of Software Maintenance: Research and Practice 14(5):293–310

Ferenc R, Beszédes Á, Tarkiainen M, Gyimóthy T (2002) Columbus—Reverse Engineering Tool and Schema for C++. 18th IEEE International Conference on Software Maintenance (ICSM'02), Montréal, Canada, 172–181.

Fischer B (1998) Specification-Based Browsing of Software Component Libraries. 13th ASE, 74–83.

Flyvbjerg B (2006) Five misunderstandings about case-study research. Qual Inq 12(2):219–245

Gall H, Jazayeri M, Krajewski J (2003) CVS Release History Data for Detecting Logical Couplings. Sixth International Workshop on Principles of Software Evolution (IWPSE'03):13–23.

Gyimóthy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Softw Eng 31(10):897–910

Hassoun Y, Johnson R, Counsell S (2004) A Dynamic Runtime Coupling Metric for Meta-Level Architectures. 8th IEEE European Conference on Software Maintenance and Reengineering (CSMR'04), 339–346

Hayes JH, Dekhtyar A, Sundaram SK (2006) Advancing candidate link generation for requirements tracing: the study of methods. IEEE Trans Softw Eng 32(1):4–19

Helm R, Maarek YS (1991) Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. Conference proceedings on Object-oriented programming systems, languages, and applications. Phoenix, Arizona, United States ACM, New York, 47–61

Hill E, Pollock L, Vijay-Shanker K (2007) Exploring the Neighborhood with Dora to Expedite Software Maintenance. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), 14–23

Jolliffe IT (1986) Principal Component Analysis. Springer, New York

Kawaguchi S, Garg PK, Matsushita M, Inoue K (2006) MUDABlue: an automatic categorization system for open source repositories. J Syst Softw 79(7):939–953

Kosara R, Healey CG, Interrante V, Laidlaw DH, Ware C (2003) Visualization viewpoints. Comput Graph and Appl 23(4):20–25

Kramer S, Kaindl H (2004) Coupling and cohesion metrics for knowledge-based systems using frames and rules. ACM Trans Softw Eng Methodol 13(3):332–358

Kuhn A, Ducasse S, Gîrba T (2007) Semantic clustering: identifying topics in source code. Inf Softw Technol 49(3):230–243

Lawrie DJ, Feild H, Binkley D (2006) Leveraged Quality Assessment using Information Retrieval Techniques. 14th IEEE International Conference on Program Comprehension (ICPC'06), 149–158.

Lee JK, Jung SJ, Kim SD, Jang WH, Ham DH (2001) Component identification method with coupling and cohesion. Eighth Asia-Pacific Software Engineering Conference (APSEC'01), 79–86.

Lee YS, Liang BS, Wu SF, Wang FJ (1995) Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow. International Conference on Software Quality, Maribor, Slovenia.

Li W, Henry S (1993) Object-oriented metrics that predict maintainability. J Syst Softw 23(2):111–122

Linstead E, Rigor P, Bajracharya S, Lopes C, Baldi P (2007) Mining Eclipse Developer Contributions via Author-Topic Models. 4th IEEE International Workshop on Mining Software Repositories (MSR'07), Minneapolis, MN, 30–33

Lo KK, Chan MK, Baniassad E (2006) Isolating and Relating Concerns in Requirements using Latent Semantic Analysis. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06), 383–396

Lormans M, Van Deursen A (2006) Can LSI help Reconstructing Requirements Traceability in Design and Test? 10th European Conference on Software Maintenance and Reengineering (CSMR'06), 47–56.

Maarek YS, Berry DM, Kaiser GE (1991) An information retrieval approach for automatically constructing software libraries. IEEE Trans Softw Eng 17(8):800–813

Maletic JI, Marcus A (2001) Supporting Program Comprehension Using Semantic and Structural Information. 23rd International Conference on Software Engineering (ICSE'01), Toronto, Ontario, Canada, IEEE, 103–112.

Marcus A, Maletic JI (2001) Identification of High-Level Concept Clones in Source Code. Automated Software Engineering (ASE'01), San Diego, CA, 107–114.

Marcus A, Poshyvanyk D (2005) The Conceptual Cohesion of Classes. 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, 133–142.

Marcus A, Sergeyev A, Rajlich V, Maletic J (2004) An Information Retrieval Approach to Concept Location in Source Code. 11th IEEE Working Conference on Reverse Engineering (WCRE'04), Delft, The Netherlands, 214–223.

Marcus A, Maletic JI, Sergeyev A (2005a) Recovery of traceability links between software documentation and source code. Int J Softw Eng Knowl Eng 15(4):811–836

Marcus A, Rajlich V, Buchta J, Petrenko M, Sergeyev A (2005b) Static Techniques for Concept Location in Object-Oriented Code. 13th IEEE International Workshop on Program Comprehension (IWPC'05), 33–42.

Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object oriented systems. IEEE Trans Softw Eng 34(2):287–300

Michail A, Notkin D (1999) Assessing software libraries by browsing similar classes, functions and relationships. IEEE International Conference on Software Engineering (ICSE'99), 463–472.

Mitchell A, Power JF (2006) A study of the influence of coverage on the relationship between static and dynamic coupling metrics. Sci Comput Program 59:4–25

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. IEEE International Conference on Software Maintenance (ICSM'00), 120–130.

Offutt J, Harrold MJ, Kolte P (1993) A software Metric System for module coupling. J Syst Softw 20 (3):295–308

Olague H, Etzkorn L, Gholston S, Quattlebaum S (2007) Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. IEEE Trans Softw Eng 33(6):402–419

Orme AM, Yao H, Etzkorn LH (2006) Coupling metrics for ontology-based systems. IEEE Softw 23:102–108

Orso A, Apiwattanapong T, Law J, Rothermel G, Harrold MJ (2004) An empirical comparison of dynamic impact analysis algorithms. IEEE/ACM International Conference on Software Engineering (ICSE'04), 776–786.

Pan Y, Wang L, Zhang L, Xie B, Yang F (2004) Relevancy based semantic interoperation of reuse repositories. 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering (FSE12), Newport Beach, CA, 211–220.

Patel S, Chu W, Baxter R (1992) A Measure For Composite Module Cohesion. International Conference on Software Engineering (ICSE'92), 38–48.

Poshyvanyk D, Marcus A (2006) The Conceptual Coupling Metrics for Object-Oriented Systems. 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, PA, 469–478.

Poshyvanyk D, Marcus D (2007) Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, 37–48.

Poshyvanyk D, Guéhéneuc YG, Marcus A, Antoniol G, Rajlich V (2007) Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. IEEE Trans Softw Eng 33 (6):420–432

Queille J-P, Voidrot J-F, Wilde N, Munro M (1994) The Impact Analysis Task in Software Maintenance: A Model and a Case Study. International Conference on Software Maintenance, 234–242.

Robillard M (2005) Automatic Generation of Suggestions for Program Investigation. Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, Lisbon, Portugal, 11–20

Rountev A, Milanova A, Ryder BG (2001) Points-to analysis for Java using annotated constraints. Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), Tampa Bay, FL, USA, 43–55.

Runeson P, Alexandersson M, Nyholm O (2007) Detection of Duplicate Defect Reports Using Natural Language Processing. 29th IEEE/ACM International Conference on Software Engineering (ICSE'07), Minneapolis, MN, 499–510.

Salton G, McGill M (1983) Introduction to Modern Information Retrieval, McGraw-Hill.

Siegel S, Castellan NJ (1988) Nonparametric Statistics for the Behavioral Sciences. McGraw-Hill, New York

Stein C, Etzkorn LH, Cox GW, Farrington PA, Gholston S, Utley DR, Fortune J (2004) A New Suite of Metrics for Object-Oriented Software. Software Audit and Metrics, 49–58.

Wang X, Zhang L, Xie T, Anvik J, Sun J (2008) An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information. 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, 461–470.

Wilkie FG, Kitchenham BA (2000) Coupling measures and change ripples in C++ application software. J Syst Softw 52:157–164

Yang HY, Tempero E, Berrigan R (2005) Detecting indirect coupling. Australian Software Engineering Conference, 212–221.

Ye Y, Fischer G (2005) Reuse-conducive development environments. Journal Automated Software Engineering 12(2):199–235

Yin RK (2003) Applications of Case Study Research. Sage Publications Inc., CA, USA

Yu P, Systa T, Muller H (2002) Predicting fault-proneness using OO metrics. An industrial case study. 6th European Conference on Software Maintenance and Reengineering (CSMR'02), 99–107

Yu Z, Vaclav R (2001) Hidden Dependencies in Program Comprehension and Change Propagation. Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Canada, 293–299.

Zhao J (2004) Measuring Coupling in Aspect-Oriented Systems. 10th IEEE International Software Metrics Symposium (METRICS'04), Chicago, USA.

Zhao W, Zhang L, Liu Y, Sun J, Yang F (2006) SNIAFL: towards a static non-interactive approach to feature location. ACM Trans Softw Eng Methodol (TOSEM) 15(2):195–226

Zou L, Godfrey MW, Hassan AE (2007) Detecting Interaction Coupling from Task Interaction Histories. 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, 135–144

Denys Poshyvanyk is an Assistant Professor at the College of William and Mary in Virginia. He received his Ph.D. degree in Computer Science from Wayne State University in 2008. He also obtained his MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006, respectively. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is member of the IEEE and ACM.

Andrian Marcus is currently an Assistant Professor at the Department of Computer Science at Wayne State University, Detroit. His research interests include software evolution, program understanding, and software visualization, in particular using information retrieval techniques to support software engineering tasks. Since 2005, he has been serving on the steering committee of the IEEE International Conference on Software Maintenance (ICSM) and he will be Program Co-Chair for the 17th IEEE International Conference on Program Comprehension (ICPC 2009) and the 26th IEEE International Conference on Software Maintenance (ICSM 2010). He is the recipient of a Fulbright Junior Research Fellowship in 1997.

Rudolf Ferenc is an Assistant Professor at the University of Szeged in Hungary. His research interests include source code analysis, modeling, measurement and design pattern recognition. He is also interested in software quality assurance and open source software development. He is Program Co-Chair of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009).

Tibor Gyimóthy is the head of the Software Engineering Department at the University of Szeged in Hungary. His research interests include program comprehension, slicing, reverse engineering and compiler optimization. He has published over 70 papers in these areas and was the leader of several software engineering R&D projects. He was the Program Co-Chair of the 21th International Conference on Software Maintenance (ICSM 2005).