Combining Conceptual and Domain-Based Couplings to Detect Database and Code Dependencies

Malcom Gethers*, Amir Aryani†, Denys Poshyvanyk*
*College of William and Mary, United States
{mgethers,denys}@cs.wm.edu
http://www.cs.wm.edu/
†RMIT University, Australia
amir.aryani@rmit.edu.au
http://www.rmit.edu.au/

***Abstract*—Knowledge of software dependencies plays an important role in program comprehension and other maintenance activities. Traditionally, dependencies are derived by source code analysis; however, such an approach can be difficult to use in multi-tier hybrid software systems, or legacy applications where conventional code analysis tools simply do not work as is. In this paper, we propose a hybrid approach to detecting software dependencies by combining conceptual and domain-based coupling metrics. In recent years, a great deal of research focused on deriving various coupling metrics from these sources of information with the aim of assisting software maintainers. Conceptual metrics specify underlying relationships encoded by developers in identifiers and comments of source code classes whereas domain metrics exploit coupling manifested in domain-level information of software components and it is independent from software implementation. The proposed approach is independent from programming language, as such it can be used in multi-tier hybrid systems or legacy applications. We report the results of an empirical case study on a large-scale enterprise system where we demonstrate that the combined approach is able to detect database and source code dependencies with higher precision and recall as compared to its standalone constituents.**

I. Introduction

Software maintenance and evolution is a particularly intricate phenomenon in case of long-lived, large-scale, database-centric, hybrid software systems written in a mix of programming languages and constantly evolving technologies. It is quite common for such systems to evolve through the years of development and maintenance efforts, a number of different developers and stakeholders, and a multitude of software artifacts, including ever growing large source code repositories and databases. One key analysis activity to support maintenance of such systems is dependency analysis or change impact analysis, which is defined as the determination of potential effects to a subject system resulting from a proposed software change [1]. The main goal of change impact analysis is to estimate the ripple change effects and prevent side effects (i.e., introduction of new bugs) of a proposed change. The scope of impact analysis includes a number of software artifacts such as requirements, design, source code, user manuals, database schemas, and test cases.

An extensive number of approaches have been developed to support impact analysis. These techniques range from those based on traditional static and dynamic analyses [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] to the recent unconventional approaches, such as those based on Information Retrieval [12], [13], [14], [15] and Mining Software Repositories [16], [17], [18], [19]. While source code analysis has been traditionally used to trace dependencies, it is becoming a challenge to apply such approaches on database centric applications with evolving database schemas and underlying programming languages [20]. This problem is becoming even more complex in case of long-lived legacy systems with obsolete programming languages or even missing source code.

In this paper we present an approach that combines conceptual and domain-based couplings to support impact analysis of source code and database dependencies. Conceptual couplings capture the extent to which domain concepts/features and software artifacts are related to each other. This information is derived using Information Retrieval (IR) based analysis of textual software artifacts that are present in the latest version of a system (e.g., comments and identifiers in a single snapshot of software) [15]. Conceptual couplings are independent from the underlying programming language or programming paradigm. Likewise, domain-based couplings are derived from the domain-level relationships embedded in software components, which are independent from software implementation [21]. As such, domain-based coupling can be used to detect dependencies even without access to the source code or the database.

The core research assumption behind our approach is that analysis of conceptual and domain-based couplings leads to better impact analysis in hybrid software systems. For change impact analysis, both conceptual and domain-based couplings have been utilized independently, however, their combined use has not been previously investigated. The proposed combination in this paper is a necessary step to evaluate a research hypothesis that such combined use of conceptual and domain-based couplings provides improvements to the accuracy of resulting impact sets.

In order to evaluate the proposed approach, we conducted an empirical study on ADempiere, an open source Enterprise

Resource Planning (ERP) system that is an excellent example of a large-scale, multi-tier, database-centric, hybrid software. In this case study we used the proposed combination to detect database and source code dependencies for a number of impact analysis tasks. The results of this empirical study demonstrated that the proposed combination of conceptual and domain-based couplings, across several cut-points, provides statistically significant improvements in accuracy over either of the two standalone techniques. For example, the combined approach reported improvements in precision values of up to 7.05% and recall values of up to 6.79% over the conceptual coupling and up to 4.22% in precision and 24.43% in recall over domain-based coupling. These results are fairly positive considering the fact that this combination does not require a complex blending of two approaches and it can be applied to systems written in a mix of programming languages with non-trivial database dependencies.

Overall, the contributions of this paper are as the following:

- We present a novel approach to detecting database and source code dependencies using a combination of conceptual and domain-based couplings.
- We present an empirical study on one of the biggest open-source enterprise systems, demonstrating how the proposed approach can be used to detect database and source code dependencies; we also compare the combined approach to the two underlying constituent approaches.

The rest of this paper is organized as follows. Section II provides a brief discussion on the background and related work to our study. Section III explains the combined approach. Section IV presents the case study, and finally Section V concludes this paper by a discussion on the future areas of investigation.

II. Background and the Related Work

The key areas of investigation in this paper are software dependencies and their relationships with two metrics of domain-based and conceptual coupling. In this section, we first describe the related work to change impact analysis and then present these metrics in more detail.

A. Change Impact Analysis

Dependency analysis and traceability analysis are the two primary methodologies for conducting change impact analysis. Commonly, software dependency analysis refers to change impact analysis of artifacts at the same level of abstraction (e.g., code to code or design to design). On the other hand, traceability analysis relates to change impact analysis of software artifacts across various levels of abstraction (e.g., code to design or requirements to test cases). Several IA approaches ranging from traditional static and dynamic analysis techniques [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] to the recent unconventional approaches, such as those based on Information Retrieval [12], [13], [14], [15] and Mining Software Repositories [16], [17], [18], [19], exist in the literature.

Coupling measures have also been applied to support change impact analysis in Object-Oriented systems [3], [22]. Wilkie et al. [22] investigated if classes with high Coupling Between Objects (CBO) metric values are more prone to ripple effects due to software changes. While CBO was determined to be a useful pointer of change-proneness overall, it was not found to be adequate to capture all possible types of changes. Briand et al. [3] investigated the use of structural coupling metrics and proposed decision models for detecting code classes that have higher probability to be changed during impact analysis. The empirical investigation into a number of structural coupling measures and their combinations showed that those coupling measures and combinations can be used to guide dependency analysis and decrease impact analysis overhead. However, the case study exposed a substantial number of change ripple effects, which are not accounted for by the structurally coupled classes.

Zimmerman et al. [19] used CVS logs for detecting evolutionary couplings between source code entities. They used association rules based on itemset mining that were formed from the change-sets and used for impact analysis. Canfora et al. [16] proposed an IR-based technique to index changed files, commit logs, and previous issue reports from the bug-tracking repositories to support impact analysis.

More recent work by Robillard [10] and Hill et al. [14] proposed tools that can assist in navigating and prioritizing software dependencies during various software maintenance tasks. For instance, the work by Hill et al. [14] relates to our approach, since it also uses textual information from the source code to identify related software entities. A systematic comparison of impact analysis approaches is not discussed here, but can be found elsewhere [7]. In the next subsections we discuss two prerequisite approaches to impact analysis in this paper: one that is based on the analysis of conceptual couplings from source code and another one based on the analysis of textual domain-based couplings.

B. Domain-Based Coupling

Identifying the relationships between domain concepts and the source code elements has been recognized as an important task in software maintenance by number of researchers [23], [24], [25]. Mapping domain entities to source code has been used for concept location [26], [27], [25], developing reverse engineering tools [28], [29], and supporting software change process [30]. In a recent work, a coupling metric, termed as *domain-based coupling*, has been proposed as a method of approximation for software dependencies using solely domain information [31]. We build our approach based on the hypothesis that this metric and the conceptual coupling can complement each other and combining them improves the accuracy of predicting software dependencies.

At the domain level we use the following terminology [31]:

- A *domain variable* is a variable unit of data that has a clear identity at the domain level.
- A *domain function* provides proactive or reactive domain-level behaviour of the system including at least one domain variable as an input or output.

- A *user interface component (UIC)* is a system component which directly interacts with users, and contains one or more domain functions.

These elements are modelled as follows: Domain variables are modelled by a finite set V. Domain functions are modelled by a finite set F, and the binary relation $USE \subseteq F \times V$ represents the relation between functions and variables as the input-output of the functions. UICs are modelled by a finite set C, and $HAS \subseteq C \times F$ represents the relation between components and functions.

Domain-based coupling has been defined as an indication of semantic similarity between UICs, and it is measured based on symmetric and asymmetric weight functions [32] . In the scope of this paper, we only use the symmetric weight function that is defined as:

$$\omega(c, c') = \frac{|c.HAS.USE \cap c'.HAS.USE|}{|c.HAS.USE \cup c'.HAS.USE|} \tag{1}$$

For the system under analysis in this paper, we extracted the information about domain variables, UICs and their relations from a part of the system business logic that is stored in the database. Although such information may not be accessible from other enterprise systems databases; however, the domain level relations might be derived from system functional specifications [33], user manuals, and even studying the behaviour of the working software.

C. Conceptual Coupling

Within a software system, identifiers account for approximately half of the source code [34]. These identifiers, which programmers use for names of classes, methods, or attributes in source code and other artifacts, contain vital information. In many cases, information encoded in identifier names provide developers with a starting point during program comprehension tasks [35]. Self-documenting identifiers, which clearly reflect the concepts that they are supposed to represent, are essential, as they decrease the time and effort required to establish fundamental comprehension for a given program task [36].

Recently, the source code analysis research community recognized the problem of extracting and analyzing conceptual information in software artifacts. A number of techniques appear in the literature which analyze conceptual information, using IR-based methods, to support software maintenance tasks. More specifically, IR methods have been used for feature location [37], [38], traceability link recovery [39], [40], software measurement [41], [42], [43], [44], and impact analysis [12], [15], [44]. Due to space limitations, we refer the interested reader to Binkley and Lawrie [45] for a detailed discussion of other applications of IR-based techniques in the context of software maintenance.

We use conceptual similarity as a primary mechanism of capturing conceptual coupling among software entities. This measure is designed to capture the conceptual relationship among documents. Formally, the conceptual similarity between software entities e_k and e_j (where e_k and e_j can be methods), is computed as the cosine between the vectors $\mathrm{v}e_k$ and $\mathrm{v}e_j$, corresponding to e_k and e_j in the vector space constructed by an IR method (e.g., Latent Semantic Indexing - LSI) [46]:

$$CSE(e_{\mathrm{k}}, e_{\mathrm{j}}) \quad = \quad \frac{ve_k^T ve_j}{|ve_k|_2 \times |ve_j|_2} \tag{2}$$

As defined, the value of $CSE(e_k, e_j) \in [-1, 1]$, as CSE is a cosine in the Vector Space Model (VSM) [47]. For source code documents, the entities can be attributes, methods, classes, files, etc. Computing class-class or file-file similarities, CSE is straightforward (e.g., e_k and e_j are substituted by a_k and a_j in the CSE formula), while deriving CSE for a pair of UICs requires additional steps. We define the conceptual similarity between two UICs as ($CSED$) $uic_k \in C$ and $uic_j \in C$ (where C is the set of UICs in software) as:

$$CSED(uic_j, uic_k) = \frac{\sum_{c_j \in uic_j} \sum_{c_k \in uic_k} CSE(c_j, c_k)}{n}, \tag{3}$$

which is the average of the similarity measures between all unordered pairs of classes from UIC uic_k and UIC uic_j. The assumption, which is used in defining CSE and $CSED$, is that if the classes of a UIC relate to each other, then the two UICs are also related. More details regarding conceptual coupling measures can be found in our preliminary work [42], [15].

To leverage the conceptual information embedded in a given release of a software systems, we first parse the source code using a developer defined level of granularity (e.g., methods, files, etc). We represent the software system as a collection of documents, referred to as a corpus. Each software artifact in the system will have a corresponding document in the corpus. We use srcML [48], which provides an XML representation of source code, for the underlying representation of the source code and textual information. Using srcML allows use to preserve the information necessary to generate the corpus (e.g., original source code contents including comments, white space, and preprocessor directives).

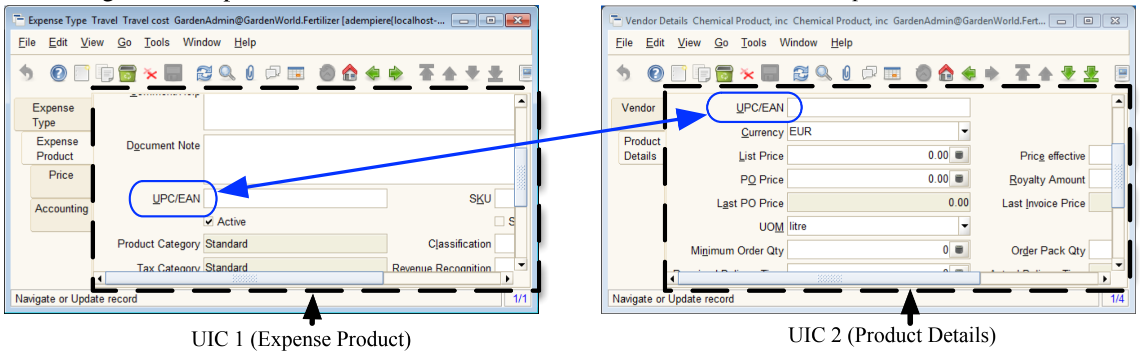
D. Software Dependencies

In our study, we use the dependency model that is derived in recent research [31] in the reverse engineering of ADEMPIERE. This model has been implemented using FAMIX [49] metamodel and Moose [50] platform. In this model the software dependencies are analysed at the source code and the database layers.

1) Source Code Dependencies: The dependency model is comprised of three main entities at the source code: Class, Attribute and Method. Classes are modelled by a finite set CLS. Attributes are modelled by a finite set ATT where the binary relation $F \subseteq CLS \times ATT$ maps them to the classes. Methods are modelled by a finite set MET where the binary relation $M \subseteq CLS \times MET$ maps them to classes. Two classes $cls, cls' \in CLS$ can have following relations:

- Returning a type: $cls.M^{-1}.R^{-1}.cls'$
- Invoking a method: $cls.M.I.M^{-1}.cls'$
- Accessing an attribute: $cls.M.A.F^{-1}.cls'$

Fig. 1: Example of two UICs in ADEMPIERE: *Product Details* and *Expense Product* tabs.



Legend: The arrows highlight a common domain variable (*EPC/EAN*) between these UICs.

For two classes $cls, cls' \in CLS$, their direct dependency has been defined as:

$$cls.D.cls' \tag{4}$$

and their indirect dependency has been defined as:

$$cls.D.D^{-1}cls' \tag{5}$$

where $D = \{M^{-1}.R^{-1}, M.I.M^{-1}, M.A.F^{-1}\}$ represents a direct relation between two classes.

2) Database Dependencies: The main entity at the database layer is the table. The set of all tables are modelled by the finite set TBL, For two tables $t, t' \in TBL$, their direct relationship has been defined as:

$$t.FK.t' \tag{6}$$

and their indirect relationship has been defined as:

$$t.FK.FK^{-1}.t' \tag{7}$$

where the binary relation $FK \subseteq TBL \times TBL$ represents the foreign key between tables.

3) Architectural Dependencies: A software component is composed of one or more classes, and two components are architecturally dependent either by a dependency between their classes or by a dependency between the tables that are accessed by these classes. More formally the binary relation $DEP \subseteq C \times CLS$ represents classes that a UIC depends on, and the relation $REF \subseteq CLS \times TBL$ represents tables that a class reads or writes to. Two components $c, c' \in C$ has been defined as architecturally dependent if and only if they are in one or more of the following relationships:

$$c.DEP.DEP^{-1}.c' \tag{8}$$
$$c.DEP.D.DEP^{-1}.c' \tag{9}$$
$$c.DEP.D.D^{-1}.DEP^{-1}.c' \tag{10}$$
$$c.DEP.REF.REF^{-1}.DEP^{-1}.c' \tag{11}$$
$$c.DEP.REF.FK.REF^{-1}.DEP^{-1}.c' \tag{12}$$
$$c.DEP.REF.FK.FK^{-1}.REF^{-1}.DEP^{-1}.c' \tag{13}$$

where Equation 8 shows a shared class between c and c', Equation 9 shows a direct dependency between their classes, Equation 10 shows an indirect dependency between their classes, Equation 11 shows a shared table between these components, and finally Equation 12 and Equation 13 represent the direct and indirect relations between their tables.

In the next example, we describe how to derive the domain-based coupling for two UICs of ADEMPIERE

E. Example

In this example, we demonstrate how we measure the domain-based coupling, conceptual coupling and architectural dependencies. *Product Details* and *Expense Product* are two UICs of ADEMPIERE that we use in this example (Figure 1).

1) Domain-Based Coupling: Expense Product (c_1) has one domain function and 17 domain variables, as follows:

$c_1.HAS = \{$ Add Product Definition of Expense Type $\}$.

$c_1.HAS.USE = \{$ Classification, Discontinued, DiscontinuedAt, UOM, UPC/EAN, TaxCategory, RevenueRecognition,... $\}$.

Product Details (c_2) contains one domain function and 23 domain variables as follows:

$c_2.HAS = \{$Edit Product Details $\}$.

$c_2.HAS.USE = \{$ LastPOPrice, DiscontinuedAt, Discontinued, OrderPackQty, PriceEffective, UOM, UPC/EAN,... $\}$.

There are 4 common domain variables between these UICs as follows:

$c_1.HAS.USE \cap c_2.HAS.USE = \{$ UOM, UPC/EAN, Discontinued, DiscontinuedAt $\}$.

and in total 36 $(17 + 23 - 4)$ variables used by either of these UICs; thus:

$$\omega(c_1, c_2) = 4/36 = 0.11$$

2) Conceptual Coupling: Expense Product (c_1) depends on two classes in the source code:

$\{org.compiere.model.MProduct,$
$org.compiere.model.X\_M\_Product\}$

Product Details (c_2) depends on two classes in the source code:

$\{org.compiere.model.X\_M\_Product\_PO,$
$org.compiere.model.MProductPO\}$

We compute $CSE(cls_j, cls_k)$ for all unordered pairs of classes, where

$$cls_j \in \{org.compiere.model.MProduct,\\ org.compiere.model.X\_M\_Product\}$$

$$cls_k \in \{org.compiere.model.X\_M\_Product\_PO,\\ org.compiere.model.MProductPO\}$$

which results in the following values for CSE, given the current example:

$$CSE(org.compiere.model.MProduct,\\ org.compiere.model.X\_M\_Product\_PO) = 0.581$$

$$CSE(org.compiere.model.MProduct,\\ org.compiere.model.MProductPO) = 0.718$$

$$CSE(org.compiere.model.X\_M\_Product,\\ org.compiere.model.X\_M\_Product\_PO) = 0.497$$

$$CSE(org.compiere.model.X\_M\_Product,\\ org.compiere.model.MProductPO) = 0.765$$

Once we have computed CSE for the set of unordered pairs of classes, we compute the average to obtain $CSED$.

$$CSED(ExpenseProduct, ProductDetails) = 0.64064$$

3) Dependencies: Analysis of the source code and the database of *Product Details* and *Expense Product* shows that there are source code and database dependencies between them. At the code layer there are three instances of indirect dependencies between the classes behind these UICs (Equation 10). At the database layer these classes read and write into two tables M_Product_PO and M_Product where these tables are connected by a direct relation (Equation 12). Hence, *Product Details* and *Expense Product* are architecturally connected.

III. Combined Approach

Our approach to combining conceptual and domain dependencies uses the union of suggestions based on conceptual and domain information (see Algorithm 1). More specifically, our approach returns the union of the set of dependencies detected by the two coupling metrics. The user specifies an initial entity and a cut point[1] for determining the number of dependencies to suggest. For a given cut point, we establish the final set of dependencies by determining the number of dependencies suggested by each individual technique as equal (or one additional element is contributed by conceptual coupling) and the cardinality of the union is equal to the specified value of the cut point. Our approach detects dependencies using the following steps:

Step 1: Select the UIC, for which we want to detect dependencies. Note that our approach starts with a given entity.

Step 2: Compute conceptual couplings between the given UIC and all other UICs to detect dependencies based on conceptual information.

Step 3: Compute domain couplings between the given UIC and all other UICs to detect dependencies based on domain information.

[1]Cut point specifies the number of predicted dependencies our approach will return to the user.

Step 4: Compute the set of dependencies from the combinations of couplings computed in steps 3 and 4. See Algorithm 1 for the exact details.

IV. Case Study

In this section we describe the design of the case study conducted to empirically assess our proposed approach. The description of our study follows the Goal-Question-Metrics paradigm [51], which includes *goals*, *quality focus*, and *context*. The *goal* of the case study is to analyze (i) whether conceptual coupling and domain-based coupling are orthogonal and (ii) whether combining conceptual and domain-based coupling metrics improves the accuracy when detecting dependencies. The *quality focus* is on ensuring improved accuracy, while the *perspective* was of a software developer inquiring about architectural dependencies in a multi-tier hybrid system.

A. Research Questions

In the context of our case study the following research questions (RQs) are addressed:

1) **RQ1**: Does conceptual and domain-based couplings detect orthogonal database and code dependencies?
2) **RQ2**: Does combining conceptual and domain-based couplings improve our ability to accurately detect database and code dependencies?

To respond to our research questions, we analyzed each coupling metric's ability to detect database and code dependencies in a multi-tier hybrid system.

B. Metrics and statistical analysis

1) Overlap: To analyze the orthogonality of the conceptual and domain-based coupling metrics (**RQ**$_1$), we used the following overlap metrics [52]:

$$correct_{m_i \cap m_j} = \frac{|correct_{m_i} \cap correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

$$correct_{m_i \setminus m_j} = \frac{|correct_{m_i} \setminus correct_{m_j}|}{|correct_{m_i} \cup correct_{m_j}|}\%$$

where $correct_{m_i}$ corresponds to the set of correctly detected dependencies by the coupling metric m_i. It is worth noting that $correct_{m_i \cap m_j}$ captures the overlap between the set of correctly detected dependencies identified by the two coupling metrics. More specifically, the metric gives an indication of what percentage of correctly detected dependencies are common to both techniques. On the other hand, $correct_{m_i \setminus m_j}$ measures the correct dependencies identified by m_i and missed by m_j. The latter metric gives an indication of how a coupling metric contributes to complementing the set of correct dependencies identified by the other metric, thus providing insight into the orthogonality between metrics.

Algorithm 1 Disjunctive combination

```
1: #Procedure to detect dependencies using disjunctive combination
2: #$entity$: initial entity to detect dependencies for
3: #$cp$ (cutpoint): number of dependencies to suggest
4: procedure DetectDepsDisj($entity, cp$)
5:     #Use domain coupling to detect $cp/2$ dependencies
6:     $DomDeps \leftarrow$ getDomCpl($entity$, $cp/2$)
7:     #Use conceptual coupling to detect $cp/2$ dependencies
8:     $ConcDeps \leftarrow$ getConcCpl($entity$, $cp/2$)
9:     #If number of detected dependencies is less than $cp$ get more dependencies
10:    if $|DomDeps \cup ConcDeps| < cp$ then
11:        #Return the set of detected dependencies
12:        return ($DomDeps \cup ConcDeps \cup$ DetectDepsDisj($entity, cp - |DomDeps \cup ConcDeps|$))
13:    end if
14:    return $DomDeps \cup ConcDeps$
15: end procedure
```

2) Precision and Recall: *Precision* and *recall*, two widely publicized information retrieval metrics [47], are used to measure the ability of the coupling metrics to detect dependencies (**RQ**$_2$). In the context of detecting dependencies, precision indicates the percentage of dependencies correctly identified, whereas recall measures the percentage of all correct dependencies in the system that are identified. These metrics are formally defined as follows:

$$recall = \frac{|cor \cap det|}{|cor|}\% \qquad precision = \frac{|cor \cap det|}{|det|}\%$$

where cor and det correspond to the sets of correct dependencies and all dependencies detected using the coupling metric, respectively.

3) Statistical Analysis: To further compare the difference in accuracy obtained using the combination of the two coupling metrics we used statistical analysis. That is, we utilized a statistical significance test to confirm that the number of correct dependencies identified by the combination of the two metrics was significantly higher than those identified using either individual metric. In other words, we compared the number of correct dependencies identified using the disjunctive combination to test the following null and alternate hypotheses:

H_0: there is no difference between the number of correct dependencies identified using the disjunctive combination.

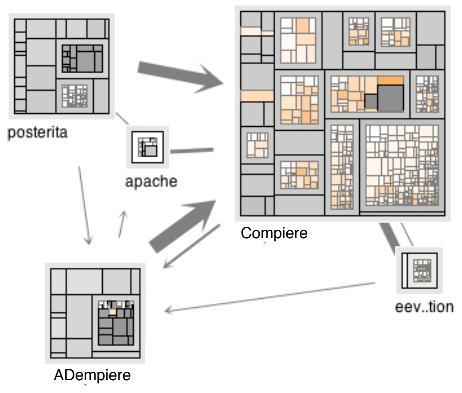
H_a: there is a statistically significant difference between the number of correct dependencies identified using the disjunctive combination.

We use the student t-test [53] and the results were intended as statistically significant at $\alpha = 0.05$.

C. System Under Analysis: ADempiere

We evaluate our hypothesis with a case study on a large scale Enterprise Resource Planning (ERP) system, called ADempiere[2]. It is composed of multiple subsystems across various domains such as accounting, asset management, sales and finance, *etc.* Such diversity enables us to limit the impact of domain specific properties (*e.g.,* complexity of business rules) on the evaluation results.

Fig. 2: High Level Architecture of ADempiere



Legend: The view is obtained by aggregating the software dependencies in the system along the package hierarchy [54].

ADempiere represents the state of art of open source systems with a multi-tier architecture and four distinct user interfaces including a Java GUI and three web interfaces. This project was forked from Compiere open source ERP that itself was created in 1999. In the last decade, ADempiere has been evolved to one the most active open source enterprise projects, and at the time of writing this paper it has the download rate of more than 1,000 times per month. Figure 2 shows the high level architecture of ADempiere. This view is created by aggregating the dependencies between source code elements, and the size of each module is proportional to the number of

[2]http://www.adempiere.com/ADempiere_ERP

TABLE I: Number of Dependencies in ADEMPIERE

	Code	IDR	DDR	ARC
PairWiseDep.	25,856	8,899	12,749	27,015

Legend: COD: Source code dependencies, IDR: Indirect database relationships, DDR: Direct database relationships, ARC: Architectural dependencies, PairWiseDep.: Number of pairwise architecturally dependent UICs

lines of code [54]. In this study, we focus on the core part of ADEMPIERE that is composed of more than 3,000 Java classes.

The user interface of ADEMPIERE is composed of windows, tabs and fields. A window has one or more tabs, and a tab includes multiple fields. In this study, we focus on tabs as the minimum user interface component with at least one domain function, that leads us to 889 UICs. We traced the dependencies between these UICs through the source code and the database layers as described in Section II-D. The outcome of the dependency analysis is presented in Table I. In summary, there are 27,015 UIC pairs connected by architectural (source code + database) dependencies.
Note that in this table, the number of pairwise dependencies represents the number of dependent UIC pairs rather than the number of individual dependencies between them.

D. Results

*1) **RQ1**: Does conceptual and domain-based couplings detect orthogonal database and code dependencies?:* We first investigate whether conceptual and domain-based couplings detect orthogonal database and code dependencies. Our focus is on determining if the two metrics actually complement one another. Given this insight, we can determine whether it will be beneficial to combine the two metrics. A scenario where augmenting the two coupling metrics is beneficial is when both metrics provide complementary sets of correct dependencies. If the two metrics identify identical or highly similar dependencies, it may not be worthwhile to combine the metrics.

With regards to the orthogonality of the two metrics, Table II presents the results of the overlap analysis. For the various cut points considered, we measure three aspects of the data. Given the set of correct dependencies identified by each metric we determine the percentage of correct dependencies (1) identified by both metrics ($correct_{conc \cap dom}$) and (2) unique to domain-based coupling metric ($correct_{conc \setminus dom}$) and (3) unique to conceptual coupling metric ($correct_{dom \setminus conc}$). As the table reveals, the percentage of correct dependencies returned by both metrics does not exceed 31%. This demonstrates that the two metrics are indeed orthogonal, given the current software system under consideration. This result provides a strong indication that combining conceptual and domain-based couplings can be beneficial. We empirically evaluate such a combination in the next section.

*2) **RQ2**: Does combining conceptual and domain-based couplings improve our ability to accurately detect database and code dependencies?:* Our primary goal is to enhance our ability to detect dependencies by using the orthogonality of coupling metrics. In this work, we consider a disjunctive approach to combining coupling metrics. Based on our finding (see Tables III, IV, V, and VI), the disjunctive approach outperforms either of the individual coupling metrics in virtually all the cases. The orthogonality of the two metrics, as discussed in the previous sub-section, appears to contribute to the results of the disjunctive combination. Our results show that the combination of conceptual and domain-based couplings improves the performance over either individual coupling metrics. For example, consider a case in Table III where cut point is 10. Conceptual and domain-based couplings in this instance yield precision values of 25.67% and 29.41% respectively, while the combination of the two increases precision to 31.99%. Comparable improvements are apparent throughout all the results. Similar results are achieved when detecting database and code dependencies. Table V indicates that, for a cut point of 100, we obtain improvements in both precision and recall when the disjunctive approach is used. In this case, conceptual and domain-based coupling yield 7.94% and 8.88% precision respectively, while the disjunctive combination gives yields precision of 9.74%. With respect to recall, conceptual and domain-based coupling returns 52.38% and 30.35% respectively. The disjunctive combination is able to correctly identify 54.78% of correct dependencies. There are a few cases where the disjunctive combination does not perform as well as one of the individual metrics (see Table IV). In most cases, this is caused by the large discrepancy between the two techniques. When there is a substantial difference in the performance of the two techniques, the technique with the lower accuracy negatively impacts the accuracy of the disjunctive approach. As discussed in Section IV-B3, statistical analysis is used to confirm that the improvements that we observe are not by chance. Our application of the statistical test determines whether the improvement in detection accuracy obtained using the disjunctive approach compared to use of each individual metric is statistically significant. Results of the statistical analysis (see Table Table VII) indicate that in most cases where we observe an improvement using the disjunctive approach, we are able to reject the null hypothesis and conclude that there is a statistical significant improvement.

E. Threats to Validity

We outline some of the threats to validity that could influence the results of our empirical case study and limit our ability to generalize our findings. We demonstrated the benefits of combining conceptual with domain-based couplings metrics to improve accuracy of impact analysis, however, our case study is performed using only one open source software system that is ADEMPIERE. Although this system is representative of a large spectrum of open-source systems in practice, to claim generalization and external validity of our results would require additional empirical evaluation on more software systems, which are also implemented in other programming languages (or even mixes of programming languages) and using different development paradigms. Yet, re-engineering such systems to

TABLE II: Overlap analysis for conceptual (Conc) and domain-based (Dom) couplings for various cut points.

	Cut Points	10	20	30	40	50	60	70	80	90	100
	$correct_{conc \cap dom}$	25.27%	25.65%	25.86%	26.90%	27.80%	28.26%	28.88%	29.84%	30.45%	30.40%
Architectural Dependencies	$correct_{conc \setminus dom}$	35.00%	37.86%	39.02%	39.24%	39.72%	40.04%	40.14%	39.92%	40.07%	40.79%
	$correct_{dom \setminus conc}$	24.67%	24.85%	24.65%	24.33%	23.90%	23.35%	28.63%	29.18%	28.54%	27.87%
	$correct_{conc \cap dom}$	10.37%	16.30%	18.86%	20.69%	22.03%	22.90%	23.70%	25.09%	25.86%	26.17%
Direct Database Dependencies	$correct_{conc \setminus dom}$	22.39%	27.00%	28.49%	29.59%	31.46%	32.63%	33.38%	34.03%	34.57%	36.11%
	$correct_{dom \setminus conc}$	40.82%	39.25%	38.50%	36.75%	34.25%	33.86%	33.01%	31.69%	30.85%	29.22%
	$correct_{conc \cap dom}$	14.20%	16.87%	19.09%	21.07%	22.87%	24.16%	26.14%	27.51%	28.98%	28.96%
Indirect Database Dependencies	$correct_{conc \setminus dom}$	38.76%	46.06%	47.90%	48.57%	49.29%	49.43%	49.53%	49.01%	48.05%	49.07%
	$correct_{dom \setminus conc}$	18.06%	17.40%	16.38%	16.05%	15.62%	14.91%	20.41%	21.28%	20.89%	20.13%
	$correct_{conc \cap dom}$	27.97%	28.18%	28.38%	29.52%	30.53%	31.00%	31.79%	32.74%	33.41%	33.40%
Source Code Dependencies	$correct_{conc \setminus dom}$	38.06%	41.35%	42.96%	43.16%	43.59%	43.84%	43.68%	43.59%	43.76%	44.49%
	$correct_{dom \setminus conc}$	25.22%	25.58%	25.18%	24.88%	24.21%	23.62%	23.12%	22.63%	21.92%	21.21%

TABLE III: Accuracy (precision (P) and recall (R)) for detecting architectural dependencies using Conceptual (Conc), Domain-based (Dom), and Combined (Comb) approaches for various cut points.

Cut Points	P(10)	R(10)	P(20)	R(20)	P(30)	R(30)	P(40)	R(40)	P(50)	R(50)	P(60)	R(60)	P(70)	R(70)	P(80)	R(80)	P(90)	R(90)	P(100)	R(100)
Conc	25.67	12.26	20.95	17.23	18.42	20.35	16.75	22.84	15.80	25.59	15.06	28.48	14.42	30.72	13.80	32.39	13.24	33.98	12.78	35.37
Dom	29.41	7.40	25.72	10.46	23.54	12.59	22.34	14.76	21.43	16.62	20.91	18.28	20.56	21.86	20.30	26.06	19.95	27.52	19.64	28.22
Comb	31.99	12.57	27.75	17.14	25.18	21.06	23.10	23.51	21.65	26.37	20.52	28.83	19.90	33.67	19.31	38.71	18.71	40.84	18.24	42.15
Comb vs. Conc	**6.32**	**0.32**	**6.81**	-0.09	**6.76**	**0.71**	**6.35**	**0.67**	**5.85**	**0.78**	**5.46**	**0.35**	**5.48**	**2.95**	**5.51**	**6.32**	**5.47**	**6.86**	**5.46**	**6.79**
Comb vs. Dom	**2.58**	**5.18**	**2.03**	**6.68**	**1.64**	**8.46**	**0.76**	**8.75**	**0.23**	**9.75**	**-0.39**	**10.55**	-0.65	**11.80**	-0.99	**12.65**	-1.24	**13.33**	-1.40	**13.93**

TABLE IV: Accuracy (precision (P) and recall (R)) for detecting direct database dependencies using Conceptual (Conc), Domain-based (Dom), and Combined (Comb) approaches for various cut points.

Cut Points	P(10)	R(10)	P(20)	R(20)	P(30)	R(30)	P(40)	R(40)	P(50)	R(50)	P(60)	R(60)	P(70)	R(70)	P(80)	R(80)	P(90)	R(90)	P(100)	R(100)
Conc	8.40	4.43	9.60	7.95	9.48	10.06	9.16	12.62	9.53	15.05	9.80	16.92	9.90	18.46	9.71	19.83	9.44	21.12	9.34	22.53
Dom	17.59	4.98	17.75	8.26	17.58	10.95	17.27	13.29	16.96	14.96	16.72	16.59	16.41	17.90	16.23	19.12	16.01	20.11	15.75	20.69
Comb	15.00	5.50	16.49	9.59	16.48	12.64	16.07	15.26	15.56	17.78	15.32	19.80	15.00	22.10	14.62	23.89	14.22	25.21	13.98	26.42
Comb vs. Conc	**6.60**	**1.07**	**6.89**	**1.64**	**7.00**	**2.59**	**6.90**	**2.65**	**6.03**	**2.73**	**5.52**	**2.87**	**5.11**	**3.64**	**4.91**	**4.07**	**4.77**	**4.08**	**4.63**	**3.89**
Comb vs. Dom	-2.59	**0.52**	-1.27	**1.32**	-1.10	**1.69**	-1.20	**1.97**	-1.39	**2.82**	-1.40	**3.21**	-1.41	**4.21**	-1.61	**4.77**	-1.79	**5.09**	-1.77	**5.73**

TABLE V: Accuracy (precision (P) and recall (R)) for detecting indirect database dependencies using Conceptual (Conc), Domain-based (Dom), and Combined (Comb) approaches for various cut points.

Cut Points	P(10)	R(10)	P(20)	R(20)	P(30)	R(30)	P(40)	R(40)	P(50)	R(50)	P(60)	R(60)	P(70)	R(70)	P(80)	R(80)	P(90)	R(90)	P(100)	R(100)
Conc	16.05	17.81	13.69	25.71	12.14	30.70	10.90	34.83	10.31	38.78	9.84	42.68	9.30	45.85	8.82	48.19	8.30	50.08	7.94	52.38
Dom	13.94	7.22	11.79	10.96	10.64	13.49	10.03	15.97	9.66	18.11	9.39	19.83	9.25	23.49	9.20	27.95	9.06	29.68	8.88	30.35
Comb	18.15	17.44	15.85	24.38	14.09	30.09	12.69	33.46	11.72	37.05	11.08	40.12	10.73	45.35	10.44	50.78	10.07	53.34	9.74	54.78
Comb vs. Conc	**2.10**	-0.37	**2.16**	-1.33	**1.95**	-0.61	**1.79**	-1.36	**1.41**	-1.73	**1.24**	-2.56	**1.43**	-0.50	**1.63**	**2.58**	**1.77**	**3.26**	**1.80**	**2.40**
Comb vs. Dom	**4.22**	**10.22**	**4.06**	**13.42**	**3.46**	**16.61**	**2.66**	**17.49**	**2.06**	**18.94**	**1.69**	**20.29**	**1.48**	**21.86**	**1.24**	**22.83**	**1.02**	**23.66**	**0.86**	**24.43**

TABLE VI: Accuracy (precision (P) and recall (R)) for detecting source code dependencies using Conceptual (Conc), Domain-based (Dom), and Combined (Comb) approaches for various cut points.

Cut Points	P(10)	R(10)	P(20)	R(20)	P(30)	R(30)	P(40)	R(40)	P(50)	R(50)	P(60)	R(60)	P(70)	R(70)	P(80)	R(80)	P(90)	R(90)	P(100)	R(100)
Conc	27.72	14.11	22.52	19.68	19.83	23.38	18.03	26.18	16.99	29.20	16.19	32.41	15.52	34.79	14.86	36.69	14.26	38.40	13.76	39.92
Dom	31.22	8.03	27.32	11.35	25.00	13.62	23.73	16.04	22.73	18.02	22.18	19.84	21.70	21.59	21.32	22.99	20.93	24.11	20.60	24.90
Comb	34.05	13.99	29.57	19.07	26.83	23.54	24.63	26.28	23.07	29.43	21.87	32.18	21.08	35.05	20.33	37.55	19.68	39.45	19.19	40.83
Comb vs. Conc	**6.33**	-0.12	**7.05**	-0.61	**7.01**	**0.16**	**6.60**	**0.11**	**6.07**	**0.23**	**5.68**	-0.24	**5.57**	**0.26**	**5.47**	**0.86**	**5.43**	**1.05**	**5.42**	**0.91**
Comb vs. Dom	**2.83**	**5.95**	**2.25**	**7.72**	**1.84**	**9.92**	**0.90**	**10.25**	**0.33**	**11.41**	-0.31	**12.34**	-0.62	**13.45**	-0.99	**14.56**	-1.24	**15.34**	-1.41	**15.93**

build benchmarks in order to evaluate approaches like this one remains a tedious and error-prone task. Thus, we make the data from our case study publicly available [3] so that other researchers can verify and perhaps even reproduce our results.

We apply an IR technique to textual information extracted from the source code of software systems to derive conceptual coupling metrics. Hence, our findings may have been impacted by the consistency of variable naming and commenting practices of ADEMPIERE software developers. The use of Latent Semantic Indexing to compute conceptual couplings is also sensitive to a set of user-defined parameters, such as pre-processing techniques and a dimensionality reduction factor, that is *k*. It is a viable risk that the results obtained by our approach are valid only for a particular set of these parameter values, that is, particular values of *k* and chosen pre-processing strategies, such as identifier splitting and stemming. To address this risk, we relied on the parameter values and pre-processing strategies that were used in our previous work [18], [13].

We measured the accuracy of impact analysis using a benchmark we created with precision and recall metrics. Our benchmark is based on the software dependencies that are derived from the FAMIX meta model and Moose technology; however, it is possible that, when creating our benchmark, we did not detect all true dependencies. Furthermore, it is possible that a different accuracy metric may generate different results; however, both these metrics are widely used and accepted in the community, including various papers on impact analysis. We

[3] http://hdl.handle.net/102.100.100/7799

TABLE VII: Statistical Analysis.

	Cut Points	P(10)	R(10)	P(20)	R(20)	P(30)	R(30)	P(40)	R(40)	P(50)	R(50)	P(60)	R(60)	P(70)	R(70)	P(80)	R(80)	P(90)	R(90)	P(100)	R(100)
Architectural	Comb vs. Conceptual	**<0.001**	0.147	**<0.001**	0.583	**<0.001**	0.065	**<0.001**	0.095	**<0.001**	0.0754	**<0.001**	0.285	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**
	Comb vs. Domain	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**0.0272**	**<0.001**	0.2849	**<0.001**	0.831	**<0.001**	0.947	**<0.001**	0.993	**<0.001**	0.999	**<0.001**	0.999	**<0.001**
Direct	Comb vs. Conceptual	**<0.001**	**0.003**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**
	Comb vs. Domain	0.999	**0.039**	0.999	**<0.001**	0.998	**<0.001**	0.998	**<0.001**	0.999	**<0.001**	0.999	**<0.001**	0.999	**<0.001**	0.999	**<0.001**	0.999	**<0.001**	0.999	**<0.001**
Indirect	Comb vs. Conceptual	**<0.001**	0.811	**<0.001**	0.987	**<0.001**	0.835	**<0.001**	0.976	**<0.001**	0.989	**<0.001**	0.999	**<0.001**	0.716	**<0.001**	**0.008**	**<0.001**	**0.002**	**<0.001**	**0.019**
	Comb vs. Domain	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**
Source Code	Comb vs. Conceptual	**<0.001**	0.657	**<0.001**	0.911	**<0.001**	0.375	**<0.001**	0.4237	**<0.001**	0.345	**<0.001**	0.638	**<0.001**	0.345	**<0.001**	0.091	**<0.001**	0.056	**<0.001**	0.089
	Comb vs. Domain	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**<0.001**	**0.018**	**<0.001**	0.219	**<0.001**	0.758	**<0.001**	0.923	**<0.001**	0.988	**<0.001**	0.998	**<0.001**	0.999	**<0.001**

also tried F-measure, which is based on precision and recall, and also noticed statistically significant improvements with our combined approach; however, we do not report the results based on F-measure in the paper because of space limitations. The accuracies of the two standalone techniques, conceptual coupling and domain-based couplings, however, seem to have low values in some cases (mostly for larger cut points, so it is to be expected); however, they are comparable to other previous results [19].

Nevertheless, the main contribution of our work is to improve accuracy by forming an effective combination. Yet, we do not claim that our combined approach would operate with equivalent improvement in accuracy on other systems, including closed source.

V. Conclusion and Future Work

The empirical evaluation on a large scale enterprise software provides support for our proposed combined approach with several conclusions in the context of dependency analysis. Overall, combining conceptual and domain-based couplings improves accuracy. The empirical findings demonstrate that in certain cases an improvement of 24.43% in recall and 7.05% in precision is achieved when conceptual and domain-based couplings are combined. Of course, some of these results and magnitude of improvements are different for database and code dependencies, however, we observe an overall improvement across different cut points for all types of dependencies. Moreover, the overall improvement in precision and recall obtained when combining the two types of couplings is statistically significant for the dataset of dependencies used in our evaluation. We conjecture that obtained improvement in accuracy for the proposed approach is, in part, due to the orthogonal nature of the correct software entities detected by the two couplings, which has been confirmed in our empirical case study.

We plan to develop and empirically corroborate other combinations of conceptual and domain-based couplings (e.g., weighed contributions of entities from conceptual and domain-based couplings based on the confidence in each source of information). One important future direction includes the addition of static and dynamic couplings as well as coupling measures based on change data, i.e., evolutionary couplings. We are also planning rigorous comparative studies with these approaches (e.g., structural and evolutionary metrics). In previous studies [18], [15], it was reported that conceptual metrics performed as well as or better than those based on structural or evolutionary metrics. Our ultimate goal is to combine all these different types of couplings to improve change impact analysis.

7 **Acknowledgments -** We would like to thank Fabrizio Perin, Dr. Mircea Lungu and Prof. Oscar Nierstrasz for providing the data and the dependency model of ADEMPIERE architecture. This work is supported by the United States NSF CCF-1218129, NSF CCF-1016868, NSF CCF-0916260 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] S. Bohner and R. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

[2] A. Beszedes, T. Gergely, J. Jasz, G. Toth, T. Gyimothy, and V. Rajlich, "Computation of static execute after relation with applications to software maintenance," in *23rd IEEE International Conference on Software Maintenance (ICSM '07)*, Paris, France, 2007, pp. 295–304.

[3] L. Briand, J. Wust, and H. Louinis, "Using coupling measurement for impact analysis in object-oriented systems," in *IEEE International Conference on Software Maintenance (ICSM'99)*. IEEE Computer Society Press, 1999, pp. 475–482.

[4] K. Gallagher and J. Lyle, "Using program slicing in software maintenance," *Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–762, 1991, gallagher91.pdf.

[5] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *25th International Conference on Software Engineering*, Portland, Oregon, 2003, pp. 308–318, law03.pdf.

[6] L. Moonen, "Lightweight impact analysis using island grammars," in *10th International Workshop on Program Comprehension (IWPC'02)*, Paris, France, 2002, pp. 219–228.

[7] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *IEEE/ACM International Conference on Software Engineering (ICSE'04)*, 2004, pp. 776–786.

[8] M. Petrenko and V. Rajlich, "Variable granularity for improving precision of impact analysis," in *17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, BC, Canada, 2009, pp. 10–19.

[9] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications(OOPSLA '04)*, Vancouver, BC, Canada, 2004, pp. 432–448.

[10] M. Robillard, "Automatic generation of suggestions for program investigation," in *Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, 2005, pp. 11 – 20.

[11] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495–509, 2003, tonella03.pdf.

[12] G. Antoniol, G. Canfora, G. Casazza, and A. Lucia, "Identifying the starting impact set of a maintenance request: A case study," in *4th European Conference on Software Maintenance and Reengineering (CSMR'00)*, Zurich, Switzerland, 2000, pp. 227–231, antoniol00.pdf.

[13] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated impact analysis for managing software changes," in *34th IEEE/ACM International Conference on Software Engineering (ICSE'12)*, Zurich, Switzerland, 2012, p. to appear 10 pages.

[14] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the neighborhood with dora to expedite software maintenance," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 2007, pp. 14–23.

[15] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[16] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *11th IEEE International Symposium on Software Metrics (METRICS'05)*, 2005, pp. 20–29.

[17] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, 1998, pp. 190 – 198.

[18] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, Beverly, Massachusetts, USA, 2010, pp. 119–128.

[19] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[20] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *30th IEEE/ACM Inernational Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 2008, pp. 451–460.

[21] A. Aryani, F. Perin, M. Lungu, A. N. Mahmood, and O. Nierstrasz, "Can we predict dependencies using domain information?" in *WCRE*, M. Pinzger, D. Poshyvanyk, and J. Buckley, Eds. IEEE Computer Society, 2011, pp. 55–64.

[22] F. Wilkie and B. Kitchenham, "Coupling measures and change ripples in c++ application software," *The Journal of Systems and Software*, vol. 52, pp. 157–164, 2000.

[23] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 1993, pp. 482–498.

[24] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings of the International Workshop on Program Comprehension*, 2002, pp. 271 – 278.

[25] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 2012.

[26] D. Ratiu and F. Deissenboeck, "How programs represent reality (and how they don't)," in *Proceedings of the Working Conference on Reverse Engineering*, Oct 2006, pp. 83 –92.

[27] ——, "From reality to programs and (not quite) back again," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2007, pp. 91 –102.

[28] S. Tilley, "Domain-retargetable reverse engineering. ii. personalized user interfaces," in *Proceedongs of the International Conference on Software Maintenance(ICSM)*, 1994, pp. 336 –342.

[29] ——, "Domain-retargetable reverse engineering. iii. layered modeling," in *Proceedongs of the International Conference on Software Maintenance (ICSM)*, 1995, pp. 52–61.

[30] K. Wong, "On inserting program understanding technology into the software change process," in *Proceedings of the Fourth Workshop on Program Comprehension*, 1996, pp. 90 –99.

[31] A. Aryani, F. Perin, M. Lungu, A. N. Mahmood, and O. Nierstrasz, "Can we predict dependencies using domain information?" in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 55–64.

[32] A. Aryani, I. D. Peake, and M. Hamilton, "Domain-based change propagation analysis: An enterprise system case study," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–9.

[33] A. Aryani, I. D. Peake, M. Hamilton, H. Schmidt, and M. Winikoff, "Change propagation analysis using domain information," in *Proceedings of the 20th Australian Software Engineering Conference (ASWEC)*. Australia: IEEE, 2009, pp. 34–43.

[34] F. Deissenboeck and M. Pizka, "Concise and consistent naming," in *13th IEEE International Workshop on Program Comprehension (IWPC'05)*, St. Louis, Missouri, USA, 2005, pp. 97–106.

[35] C. Caprile and P. Tonella, "Nomen est omen: Analyzing the language of function identifiers," in *6th IEEE Working Conference on Reverse Engineering (WCRE'99)*, Atlanta, Georgia, USA, 1999, pp. 112–122.

[36] G. Antoniol, Y.-G. Gueheneuc, E. Merlo, and P. Tonella, "Mining the lexicon used by programmers during software evolution," in *23rd IEEE International Conference on Software Maintenance (ICSM'07)*. Paris, France: IEEE Computer Society Press, 2007, pp. 14–23.

[37] D. Poshyvanyk, Y. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[38] D. Poshyvanyk and D. Marcus, "Combining formal concept analysis with information retrieval for concept location in source code," in *15th IEEE International Conference on Program Comprehension (ICPC'07)*, Banff, Alberta, Canada, 2007, pp. 37–48.

[39] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970 – 983, 2002.

[40] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artefact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 16, no. 4, 2007.

[41] A. Marcus and D. Poshyvanyk, "The conceptual cohesion of classes," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, 2005, pp. 133–142.

[42] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, Philadelphia, PA, USA, 2006, pp. 469 – 478.

[43] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides, "Modeling class cohesion as mixtures of latent topics," in *25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Alberta, Canada, September 20-26 2009, pp. 233–242.

[44] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *26th IEEE International Conference on Software Maintenance (ICSM'10)*, Timişoara, Romania, 2010, pp. 1–10.

[45] D. Binkley and D. Lawrie, "Maintenance and evolution: Information retrieval applications," in *Encyclopedia of Software Engineering*, P. A. Laplante, Ed. Taylor & Francis, 2010, pp. 454–463.

[46] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse Processes*, vol. 25, no. 2&3, pp. 259–284, 1998.

[47] R. A. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[48] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *11th IEEE International Workshop on Program Comprehension (IWPC'03)*. Portland, OR: IEEE-CS, 2003, pp. 134–143.

[49] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*. IEEE Computer Society Press, 2000, pp. 157–167. [Online]. Available: http://scg.unibe.ch/archive/papers/Tich00bRefactoringMetamodel.pdf

[50] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*. New York NY: ACM Press, 2005, pp. 1–10, invited paper. [Online]. Available: http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf

[51] V. R. Basili, G. Caldiera, and D. H. Rombach., *The Goal Question Metric Paradigm*. John W & S, 1994.

[52] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *18th IEEE International Conference on Program Comprehension (ICPC'10)*, Braga, Portugal, 2010, pp. 68–71.

[53] M. D. Smucker, J. Allan, and B. Carterette, "A comparison of statistical significance tests for information retrieval evaluation," in *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, ser. CIKM '07. New York, NY, USA: ACM, 2007, pp. 623–632. [Online]. Available: http://doi.acm.org/10.1145/1321440.1321528

[54] M. Lungu and M. Lanza, "Softwarenaut: Exploring hierarchical system decompositions," in *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*. Los Alamitos CA: IEEE Computer Society Press, 2006, pp. 351–354.