JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 0000; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.1598

Predicting Dependencies Using Domain-Based Coupling

Amir Aryani[1*], Fabrizio Perin[2], Mircea Lungu[2], Abdun Naser Mahmood[1] and Oscar Nierstrasz[2]

[1] *RMIT University, Australia.*
[2] *Software Composition Group, University of Bern, Switzerland.*

SUMMARY

Software dependencies play a vital role in program comprehension, change impact analysis and other software maintenance activities. Traditionally, these activities are supported by source code analysis; however, the source code is sometimes inaccessible or difficult to analyse, as in hybrid systems composed of source code in multiple languages using various paradigms (*e.g.,* object-oriented programming and relational databases). Moreover, not all stakeholders have adequate knowledge to perform such analyses. For example, non-technical domain experts and consultants raise most maintenance requests; however, they cannot predict the cost and impact of the requested changes without the support of the developers.
We propose a novel approach to predicting software dependencies by exploiting the coupling present in domain-level information. Our approach is independent of the software implementation; hence, it can be used to approximate architectural dependencies without access to the source code or the database. As such, it can be applied to hybrid systems with heterogeneous source code or legacy systems with missing source code. In addition, this approach is based solely on information visible and understandable to domain users; therefore, it can be efficiently used by domain experts without the support of software developers. We evaluate our approach with a case study on a large-scale enterprise system, in which we demonstrate how up to 65% of the source code dependencies and 77% of the database dependencies are predicted solely based on domain information.

Received . . .

KEY WORDS: Domain-based coupling, Architectural dependencies, Database dependencies, Source code analysis, Program comprehension

PREPRINT. Final version published online at DOI:10.1002/smr.1598

1. INTRODUCTION

When software maintainers change a software entity, they have to search for other related entities and update them accordingly. This is not a trivial task, and many bugs are introduced by programmers who fail to properly propagate changes [1]. Knowledge of software dependencies is vital to many change impact analysis methods and other maintenance activities [2, 3, 4, 5].

Source code analysis can be used to trace dependencies [6]; however, it is not an easy approach to apply in many situations. As software systems become more interoperable, it is common to see hybrid systems composed of multiple programming languages (*e.g.,* C++ and Python). It is often impractical to trace source code dependencies within these systems using conventional code analysis tools targeting a single language. The other difficulty in implementing existing code analysis tools is the required level of technical expertise that is beyond the knowledge of typical programmers.

*Correspondence to: Journals Production Department, John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, UK.

Therefore, it is common practice in enterprise software environments for developers to trace the dependencies and the change propagation by manually searching the source code.

A large majority of enterprise software systems are derived from domains where requirements are uncertain and are likely to change during the software's lifetime [7]. In these domains, the domain experts are the primary source of information for evaluating requirements [8]. These domain experts drive software evolution by continuously asking for new functionality or requesting changes to existing ones. Unfortunately, domain experts are in a poor position to estimate the impact of the changes that they request because they do not have inside knowledge of the software dependencies.

Enterprise software systems are constructed to model business domains [7]. Therefore, it is reasonable to expect that real-world dependencies be reflected in the software itself. Consequently, we hypothesize that *software dependencies can be predicted by exploiting domain information*.

In this paper, we propose a novel approach to predicting software dependencies based on the notion of domain-based coupling [9] which is derived from the domain-level relationships between software components. Although the proposed method returns the probability of dependencies existing between components rather than the actual dependencies, it offers software maintainers the following benefits:

1. It is source code independent, so it can be used where the software source code is not available or not supported by code analysis tools. For this reason it can also assist in tracing inter-system dependencies in hybrid systems with heterogeneous source code.
2. It solely relies on domain information, thus allowing non-technical domain experts (*e.g.,* consultants, subject matter experts and managers) to predict the impact of software changes without the support of developers. Such a prediction can assist software maintainers by improving the change management process.
3. The proposed approach is based on the software domain-level model; hence, we envisage that this approach can be used to evaluate the complexity of software implementation with respect to the software domain-level relationships.

We evaluate our approach with a case study of a large-scale enterprise system, called ADEMPIERE, where we demonstrate how domain information can be used to identify dependencies in the source code and database layers. ADEMPIERE[†] is an Enterprise Resource Planning (ERP) software package that integrates internal and external management information across an entire organisation. We have chosen this system as a case study, as it is a large, complex, multi-language system developed over many years, with a large user base. Our results shows that we can approximate architectural dependencies with more than 70% accuracy. In this study, we report how efficiently domain-based coupling can assist software maintainers in the following scenarios:

Searching for source code dependencies: Suppose a software maintainer has no access to source code analysis tools. Using software domain information, how accurately can she predict the existence of source code dependencies between various parts of a software system?

Searching for database relationships: Some business constraints and relationships are defined and managed at the data layer. These relationships may or may not be visible at the source code level [3, 2], or can be difficult to analyse as in legacy databases. How accurately can a domain expert predict such relationships without analysing the database?

Searching for architectural dependencies: When a domain expert needs to estimate the impact of a change to a user interface component such as a data entry screen, she needs to predict which other components might be affected due to architectural dependencies. How accurate can such a prediction be using solely domain information?

In summary, the contributions of this paper are as follows:

- We refine our previously defined domain-based coupling [9] and we extend our previous method of selecting the highly coupled components with the help of an automated clustering technique.

[†]`http://www.adempiere.com`

DOI: 10.1002/smr.1598

- We formally define architectural dependencies and propose a model to trace dependencies among source code, database and user interface components.
- We present an empirical study of one of the biggest open source enterprise systems, demonstrating how domain-based coupling can be used to predict source code and database dependencies.

A shorter version of this paper has been published in the proceedings of the 18th Working Conference on Reverse Engineering [10]. This paper provides the following additional information: advanced details on the implementation of domain-based coupling (Section 2.4), details on the implementation of the source code based dependency model (Section 3.4), an expanded evaluation section (Section 4), an analysis of the impact of granularity on the results (Section 4.10), and general considerations on the applicability of the domain-based coupling (Section 5). In addition, this paper extends the discussion of dependencies in ADEMPIERE (Section 3.5).

The rest of this paper is organised as follows: Section 2 describes the domain-based coupling analysis. Section 3 presents the dependency analysis. Section 4 demonstrates the evaluation results. Section 5 describes the applicability of the domain-based approach to various software types. Section 6 discusses the threats to the validity of our findings. Section 7 presents the related work, and finally, Section 8 concludes this paper with a discussion about future areas of investigation.

2. DOMAIN-BASED COUPLING ANALYSIS

Domain information can reveal relationships among user interface components (UICs) [11]. In this section, we describe how the domain-based coupling [9] derived from software domain information can be used to predict dependencies between the user interface components.

We use the following terminology when we talk about the domain model of a system:

- A *domain variable* is a variable unit of data which has a clear identity at the domain level.
- A *domain function* provides proactive or reactive domain-level behaviour of the system which includes 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.

For example, in a business software system, a data entry form is considered a UIC, the entry and editing of business information are domain functions, and the data fields visible on the form are domain variables.

2.1. Notations and Definitions

Most of this section quotes our earlier works [9, 11] with the exception of new definitions of the number of common variables (Definition 2), and revised definitions of the domain-based coupling graph (Definition 3).

We adopt the following conventions in this work. For $R, Q \subseteq A \times A$, we denote by $R.Q$ their composition, *i.e.,* $x.R.Q.y$ iff $\exists z : x.R.z \wedge z.Q.y$. We also denote by R^{-1} the inverse of R and by ID the identity relation.

Moreover we abbreviate $x.R = \{y | x.R.y\}$. We visualise relations as graphs, denoting by $G = (V, E, l)$ the graph G with vertices V, edges $E \subseteq V \times V$ and labels $l : E \to L$ for some label set L.

If L is a finite set of relation labels and $l_R \in L$ the name of R for any $R \in X$, then we define $REL(A, X)$ to be the labelled directed graph $REL(A, X) = (V, E, l)$ with $V = A, E = \bigcup_{R \in X} R$ such that: $(v, v') \in E$ *and* $l(v, v') = l_R$ *iff* $v.R.v'$ *for some* $R \in X$.

The three key element types are modelled as follows:

- *Domain variables* are modelled by a finite set V, called *variable symbols*.
- *Domain functions* are modelled by a finite set F, called *function symbols*, and the binary relation $USE \subseteq F \times V$ represents the relation between functions and variables as the input-output of the functions.

DOI: 10.1002/smr.1598

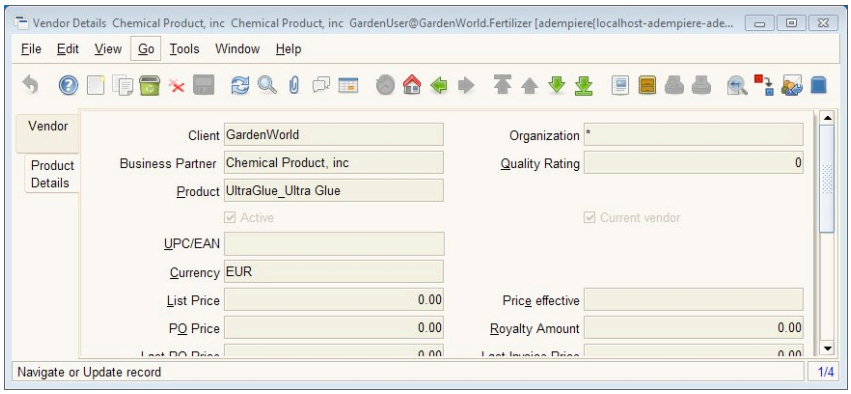


Figure 1. ADEMPIERE: The Vendor Details UIC

- *UICs* are modelled by a finite set C called the *component symbols*, and $HAS \subseteq C \times F$ represents the relation between components and functions.

For the rest of the paper, and without loss of generality, we assume that the system under analysis (SUA) is fixed, that is, V, F and C are fixed and so are their REF, USE and HAS relations.

Definition 1
The *conceptual connection* relation $CNC \subseteq C \times C$ is defined by

$$CNC = HAS.USE.USE^{-1}HAS^{-1}$$

The domain-based coupling between two components is derived from shared domain variables, based on the following measurements:

Definition 2
The number of common variables among two UICs is modelled by the function $\vartheta : C \times C \to \mathbb{R}$ where

$$\vartheta(c, c') = |c.HAS.USE \cap c'.HAS.USE|$$

Note that the definition of common domain variables is symmetric, i.e., $\vartheta(c, c') = \vartheta(c', c)$.

Definition 3
The *domain-based coupling graph* of a SUA is the symmetric weighted graph $G = (C, CNC\backslash ID, \omega)$ where coupling weight function $\omega : C \times C \to [0..1]$ is

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

It turns out that it is practically useful to weight domain relationships by their level of sharing domain variables. A threshold t can be used to select relevant coupling by their weight $\omega \geq t$. In the following examples, we demonstrate how to derive domain-based coupling from UICs of ADEMPIERE, and then how to approximate dependencies from that coupling.

2.2. Example 1

In ADEMPIERE, *Vendor Details* (Figure 1) and *Import Product* are the UICs which we use in this example. *Vendor Details* (c_1) has 2 domain functions, and in total 25 domain variables, as follows:

$$c_1.HAS = \{ \text{Edit Vendor, Edit ProductDetails} \}.$$

$$c_1.HAS.USE = \{ \text{DeliveryTime, BusinessPartner, CostPerOrder, Currency, Vendor, Manufacturer,...} \}.$$

DOI: 10.1002/smr.1598

Import Product (c_2) contains one domain function and 42 domain variables as follows:

$c_2.HAS = \{$ Import Products $\}$.

$c_2.HAS.USE = \{$ CostPerOrder, PriceEffective, Weight, BusinessPartner, SKU, UOM, Royalty,... $\}$.

There are 18 common domain variables between these UICs as follows:

$c_1.HAS.USE \cap c_2.HAS.USE = \{$ BusinessPartner, CostPerOrder, Currency, Discontinued, DiscontinuedAt, ListPrice, Manufacturer, MinOrderQty, OrderPackQty, PartnerCategory, PartnerProductKey, POPrice, PriceEffective, Product, PromisedDeliveryTime, Royalty, UOM, UPC/EAN $\}$.

and in total 49 $(42 + 25 - 18)$ variables used by either of these UICs; thus:

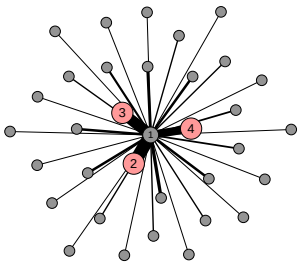
$$\vartheta(c_1, c_2) = 18$$
$$\omega(c_1, c_2) = 18/49 = 0.37$$

The next section demonstrates how to create a weighted graph from CNC relations of *Vendor Details*.

2.3. Example 2

Now that we have explained the domain definitions, let us demonstrate how to use them for predicting dependencies. Imagine a domain expert who considers asking for an enhancement to *Vendor Details* (c_1). Then given the domain information of ADEMPIERE, she can derive common domain variables (ϑ) among c_1 and other UICs similar to what was described in the previous example.

Figure 2 (next page) shows there are 33 UICs for which the coupling weight with c_1 is greater than a given threshold $\omega \geq 0.5$. The selected threshold is applied to avoid weak results which do not likely lead to any architectural dependencies. This also reduces the density of the resulting domain-based coupling graph and makes it more readable. The results are illustrated (Figure 2) as a weighted graph where the edge width is proportional to ω, and edge length is proportional to $1/\omega$, *i.e.,* the stronger the coupling weight, the thicker is the edge and the closer the node to the center (c_1).



Legend: Nodes represent UICs and edges represent domain-based coupling. The tagged nodes are (1) Vendor Details, (2) Import Products, (3) Spare Parts and (4) Product Planning. Node size has no implication, but edge width is proportional to ω and edge length is proportional to $1/\omega$. For readability, the graph only contains $c_1.CNC$, excluding edges between other nodes.

Figure 2. Vendor Details: Domain-Based Coupling Graph

The top 3 closest UICs are: *Import Products* (c_2), *Spare parts,* (c_3) and *Product Planning* (c_4), where the coupling weight values are 0.37, 0.32 and 0.25 respectively. Investigating the source code shows that all three UICs are connected to *Vendor Details* by source code dependencies.

2.4. Implementation

The ADEMPIERE user interface is composed of three major elements: *data fields*, *tabs* and *windows*. Each window is composed of one or more tabs, and each tab has multiple data fields and provides one or more domain functions.

DOI: 10.1002/smr.1598

Both windows and tabs provide one or more domain functions and they interact with the end user; therefore, they are qualified as UICs. In the rest of this paper, we discuss the relationships between ADEMPIERE UICs at the macro level and we refer to a window as a UIC. It is only in Section 4.10 that we examine the micro level granularity of the UICs and discuss the impact of the granularity of UICs on the evaluation results.

In prior works [11, 9], the system functional specification document and user manuals have been used as the source of information about the software domain-level elements. Domain experts use these sources to derive the relationships between UICs and create the domain-based coupling graph based on the following manual process:

- **Step1 - Identifying UICs:** Any software component that interacts with the end user and has one or more domain functions is a UIC. There are multiple sources for deriving the list of UICs including system user manual, help documents and the software menu. It is common for an enterprise application that its major UICs will be accessible through the software menu. Although in most systems the visibility of the items in the software menu is limited based on users privileges, the complete list is often available to the system administrator. Therefore, the list of UICs can be derived from the working software menu using administrator privileges. This function is platform independent, and web-based enterprise systems have often similar menus to desktop applications, *i.e.,* online accounting systems, banking systems and facility management systems.
- **Step2 - Identifying related domain variables:** For most enterprise systems the domain variables are the data fields that are visible on the UICs. Domain experts review the functionality of UICs by interacting with the running software or reading the user manual, and answering the following question for individual units of data: *Is the data understandable purely with domain knowledge?* The answer to this question will indicate whether a domain user who has no familiarity with the architecture and the source code of the given application can still understand the meaning and purpose of the given data within the domain. If a data field is related to a particular system behaviour such as *Screen ID* or *Last Modified Record*, then this is a system variable and it will be excluded from the list of domain variables. The list of the associated domain variables to each UIC can be recorded with a generic tool like a spreadsheet, then the CNC relationships will be derived automatically from this information using a script or a spreadsheet's macro.
- **Step3 - Creating the domain-based coupling graph:** The aim of this step is to create a weighted graph that represents the strength of the CNC relationships between UICs and identify the clusters of highly coupled components. The nodes of the graph will be UICs, the edges will be CNC relationships between UICs and the weight of each edge is ω, the coupling weight function. There are a number of graph analysis tools that can be used to automatically analyse and visualise a weighted graph such as the open source network analysis tool GEPHI [12].

Although the described process works for most enterprise systems, the required labour for collecting the domain information by domain experts has been a drawback in this approach. One of the resources about the domain information is the system database. In the case of ADEMPIERE, the relationships between data fields, tabs and windows are stored in a part of the ADEMPIERE's database called *application dictionary*. We took advantage of this part of ADEMPIERE's database to automatically derive the list of UICs and their associated domain variables using the following steps:

1. Use a SQL script to extract the list of windows from the application dictionary.
2. Extract the list of data fields in ADEMPIERE.
3. Review these data fields and exclude the fields which do not contain domain information. The remainder are considered to be domain variables. For example: *Tax Group*, *Bank Account*, *Asset Number* are domain variables whilst *Help* and *Search Key* are not domain variables, and we exclude them from the list. The result of our domain analysis yields 348 UICs and $2,359$ domain variables, leading to $18,451$ pairwise CNC relationships.

DOI: 10.1002/smr.1598

4. Use an SQL script to extract the relationships between UICs and domain variables from the application dictionary, then transform these relationships into the domain-based coupling graph (Definition 3).

Please note that although we have used application dictionary for this study, similar results can be derived from the three steps manual process and without using application dictonary.

2.5. Expectation Maximisation Clustering

In Section 2.3, we discussed using a threshold value for domain-based coupling to identify highly coupled components. Previously, the threshold value has been selected manually based on the system characteristics like distribution of the coupling values, or by graph visualisation [11]. However, the manual approach is subject to human errors and does not scale for large data sets. In order to address this limitation, in this work we use a clustering technique to automatically identify highly coupled components.

The aim of clustering is to group a given set of objects so that similar objects are grouped together and dissimilar objects are kept apart. There are many different multi-dimensional clustering techniques [13]. In this paper, we have used a statistical clustering technique called Expectation Maximization (EM)[‡] since it can automatically find the optimum number of clusters [14].

The main idea behind EM is to fit the parameters of a distribution model by using training data. The EM algorithm assigns a probability distribution to each instance of the *number of common variables* (ϑ), which indicates the probability of the instance belonging to each of the generated clusters. In this study, the training set is the same as the dataset and there is no test dataset since this is an unsupervised technique. In Section 4.8, we demonstrate how EM clustering improves the precision of identifying dependencies.

3. DEPENDENCY ANALYSIS

ADEMPIERE has been designed in such a way that a developer can extend the system by touching as little code as possible. Whenever a new table is added to the database, the required Java code is automatically generated.

Most domain-level relations are managed at the data layer. As a consequence, traditional coupling metrics fail to capture the domain-level relationships between these classes. Moreover, the database contains important information about the architectural dependencies in the system. We therefore need to develop a model which is capable of expressing dependencies both at the source code and at the database layers.

In this section, we present two general models for representing a system and its architectural relationships based on the analysis of the source code and the database. We also explain how we populated our model in the particular case of ADEMPIERE.

3.1. Source Code Dependencies

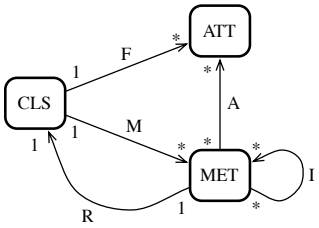
The main wellspring of architectural relations is the source code. At the source code level our analysis models three key entities and their associated relations. These entities are independent of the programming language, as long as it is object-oriented:

- *Classes* are represented by a finite set CLS.
- *Attributes* are represented by a finite set ATT. The binary relation $F \subseteq CLS \times ATT$ maps attributes to the containing classes.
- *Methods* are represented by the finite set MET. The binary relation $M \subseteq CLS \times MET$ maps methods to the classes that contain them.

[‡]EM can be a supervised technique when it is used to build a classifier. However, in this work we use EM clustering which is an unsupervised technique.

DOI: 10.1002/smr.1598

In addition, the relation $R \subseteq MET \times CLS$ expresses the return types of methods (NB: we allow $Void \in CLS$ to model methods that return void), $I \subseteq MET \times MET$ represents method invocations, and $A \subseteq MET \times ATT$ represents the accesses of methods to attributes. These relationships are illustrated in Figure 3.



Legend: CLS: classes, ATT: attributes, MET: methods

Figure 3. Source Code Elements And Relations

Two classes $cls, cls' \in CLS$ can have following relationships:

$$cls.M^{-1}.R^{-1}.cls' \tag{1}$$

$$cls.M.I.M^{-1}.cls' \tag{2}$$

$$cls.M.A.F^{-1}.cls' \tag{3}$$

Where Equation 1 shows cls is the return type of cls', Equation 2 shows a method of cls invokes a method of cls', and Equation 3 shows a method of cls accesses an attribute of cls'.

Definition 4
A direct relation between two classes is defined as $D = \{M^{-1}.R^{-1}, M.I.M^{-1}, M.A.F^{-1}\}$. For two classes $cls, cls' \in CLS$, we denote that cls is directly dependent on cls' by $cls.D.cls'$

Definition 5
For two classes $cls, cls' \in CLS$, we denote that cls is indirectly dependent on cls' by $cls.D.D^{-1}cls'$

3.2. Database Relationships

A significant part of a system's business logic is incorporated in the database relationships, and these relationships complement those that are visible at the source code level.

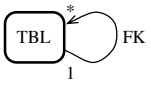


Figure 4. Database Table With The Foreign Key Relation

The main type of entity that we model at the database level is the table, and we denote the set of all the tables by TBL. The binary relation $FK \subseteq TBL \times TBL$ maps tables to tables based on the foreign keys. Figure 4 illustrates this relationship.

As in the case of source code, we define both direct and indirect relationships in the database:

Definition 6
Given two tables $t, t' \in TBL$, we say that t has a direct relation to t' if and only if $t.FK.t'$.

Definition 7
Given two tables $t, t' \in TBL$, we say that t has indirect relation to t' if and only if $t.FK.FK^{-1}.t'$

While foreign key relations among tables are there to model a specific aspect of the domain, indirect relations between tables should suggest how different concepts are bound together.

DOI: 10.1002/smr.1598

3.3. Architectural dependencies

Two components are considered to be architecturally dependent either by direct or indirect dependencies between the classes behind them, or by direct or indirect relationships between the tables accessed by these classes.

Figure 5 shows the relations between the Components (C), Classes (CLS) and Tables (TBL) of ADEMPIERE. These elements are related by $DEP \subseteq C \times CLS$ which represents classes that a UIC depends on, and $REF \subseteq CLS \times TBL$ which represents tables that a class reads or writes to.

C: components, CLS: classes, TBL:tables

Figure 5. Relationships Between Software Elements

Definition 8
For two components $c, c' \in C$, we say that c has an architectural dependency to c' if and only if they are in one or more of the following relationships:

$$c.DEP.DEP^{-1}.c' \tag{4}$$
$$c.DEP.D.DEP^{-1}.c' \tag{5}$$
$$c.DEP.D.D^{-1}.DEP^{-1}.c' \tag{6}$$
$$c.DEP.REF.REF^{-1}.DEP^{-1}.c' \tag{7}$$
$$c.DEP.REF.FK.REF^{-1}.DEP^{-1}.c' \tag{8}$$
$$c.DEP.REF.FK.FK^{-1}.REF^{-1}.DEP^{-1}.c' \tag{9}$$

This definition describes all direct and indirect dependencies through classes or tables. Equation 4 defines a connection between two components based on shared classes. Equation 5 and Equation 6 consider respectively direct and indirect dependencies between classes to connect the components depending on them. Equation 7 defines a connection between two components based on their shared database tables. Equation 8 and Equation 9 consider direct and indirect dependencies between database tables which connect two components.

3.4. Implementation

The analysis on ADEMPIERE has been performed using the Moose [15] platform for software and data analysis. One of the Moose core components is the FAMIX [16] meta-model which describes the static structure of object-oriented software systems. This abstract representation contains all the elements composing a software oriented system (*i.e.,* classes, methods, attributes, namespaces *etc.*) together with all the associations among them (*i.e.,* inheritances, invocations, accesses *etc.*). The source code dependencies described in subsection 3.1 are computed directly from the FAMIX core. Since ADEMPIERE is not simply implemented on Java but it also rely on a database, to perform the analysis on this system we first needed to extend *FAMIX* with a meta-model for relational databases. This extension is similar to the one proposed by Marinescu [17] but with more detailed relations between the software entities and the relational elements.

Figure 6 shows the subset of the extended meta-model where the new elements modeling relational databases are indicated in bold.

The entities modeling relational databases are self explanatory, more interesting are the relations between meta-model extensions and the meta-model for object-oriented systems:

- A class that *maps* a table is a class that represents a table at the source code level, *e.g.,* Enterprise Entity beans, *idem* for a *map* between a class attribute and a table column.
- The relation *access* represents class methods accessing database tables. The access can be made directly (*e.g.,* using the *java.sql* package) or through a framework (*e.g.,* Hibernate).

DOI: 10.1002/smr.1598

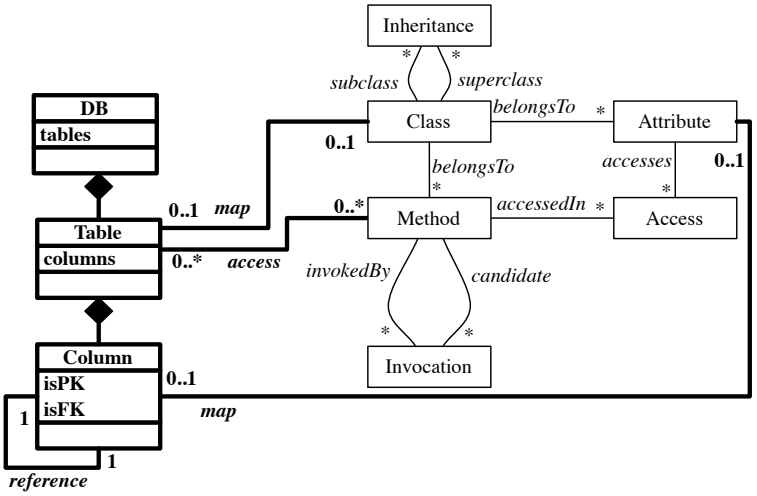


Figure 6. Extended version of the FAMIX Meta-Model including a meta-model for relational databases

- The relation *reference* represents connections among table columns established using a foreign key constraint.

These modifications to the FAMIX meta-model have been implemented in MooseJEE [18], an extension of the Moose [15] software analysis platform.

The entities and relations of the meta-model just described are generic and independent from any kind of platform or software to analyse. What is platform-related is the place where the information we need is stored. Consequently the fact extractor may need to be changed from one system to another.

3.5. Dependency Analysis in ADEMPIERE

The first step we took to analyse ADEMPIERE was to import the code and the database into the unified meta-model described in subsection 3.4. Once the two models were populated, we needed to extract the mapping between UI components and classes, and between classes and tables. In the case of ADEMPIERE, the mappings between the UI components and the classes can be found in the *Application Dictionary* as discussed in Section 2.4. The *Application Dictionary* is used by a code generator to build the application structure and the relations among the various application components. So the relations between the source code elements and the UI elements are explicitly encoded in the database. We extracted this information by querying the database.

The mapping between the database tables and the source code is based on a naming convention. For each domain-related table specified in the *Application Dictionary*, a Java class named "*X_tableName*" and a Java Interface named "*I_tableName*" are generated by ADEMPIERE. For example, for the table called *A_Asset_Acct* a class called *X_A_Asset_Acct* and an interface called *I_A_Asset_Acct* will be generated.

By creating this mapping between classes and tables we determined that 76 tables in the database are not related to any class. 65 of them are tables used for the language localization, 9 are materialized views and 2 are used for logging purposes.

The architecture of ADEMPIERE only contains one-to-one mappings, hence the relation *map* between classes and tables in our meta-model can represent all the relations we need. Other kinds of mappings should be modeled differently. Once we had these mappings we were able to compute the dependencies between the components based on architectural relationships.

DOI: 10.1002/smr.1598

4. EVALUATION

In this section, we provide empirical evidence of the usefulness of domain-based coupling in approximating architectural dependencies. We answer the following questions in this evaluation:

RQ1: How accurately can we predict dependencies between UICs at the source code level?

RQ2: How accurately can we predict relationships between UICs at the database layer?

RQ3: How accurately can we predict architectural dependencies between UICs?

RQ4: What is the impact of granularity of UICs on the prediction results?

Note that all the predictions in this evaluation are performed based on only domain information and domain-based coupling between UICs.

4.1. Evaluation Setup

For a given UIC, $c \in C$, we test the query $q = (c, E, AN)$ where the expected outcome $E \subseteq C$ is the set of UICs that have architectural dependencies to c, and the returned answer

$$AN = \{c_i | c_i \in C, \vartheta(c, c_i) > 0\}$$

is the set of UICs that are coupled with c at the domain level. We describe the outcome of such a query as follows:

TP =$|E \cap AN|$ shows the number of correctly identified dependent components.
TN =$|C \backslash \{AN \cup E\}|$ shows the number of correctly identified independent components.
FP =$|AN \backslash E|$ shows the number of incorrectly predicted dependent components.
FN =$|E \backslash AN|$, shows the number of incorrectly predicted independent components.

For the system under analysis, we measure the percentage of the queries with at least one correct answer using the *feedback* (FB) metric:

$$FB = \frac{|\{q | q = (c, E, AN), c \in C, E = 0 \vee TP_q > 0|\}}{|\{q | q = (c, E, AN), c \in C\}|}$$

We use the well-known definitions of *precision* (P_q) and *recall* (R) to evaluate the outcomes of a given query:

$$P_q = \frac{TP_q}{TP_q + FP_q} \qquad R_q = \frac{TP_q}{TP_q + FN_q}$$

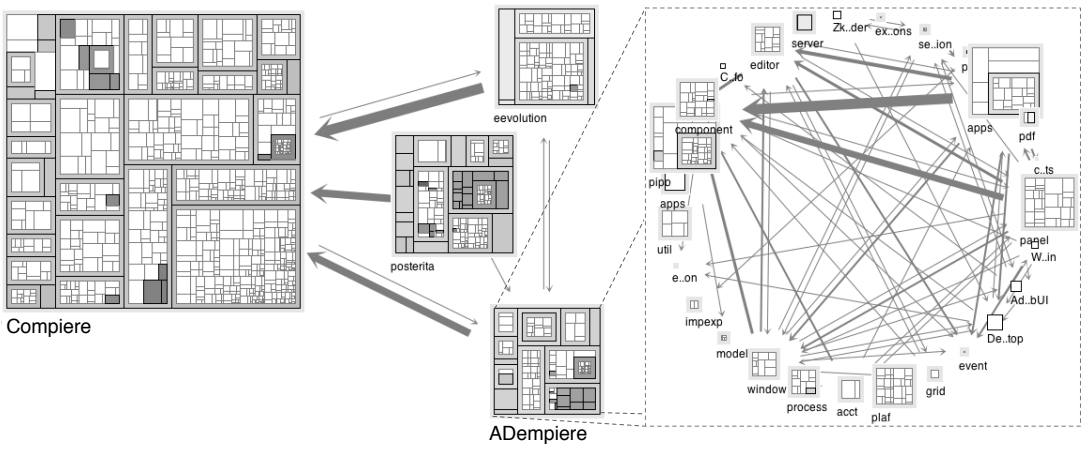
In addition, we report on the F-measure (F_1) which is the harmonic mean of the precision and recall:

$$F_{1q} = 2 \times \frac{P_q \times R_q}{P_q + R_q}$$

Precision and *recall* only evaluate TP, in order to describe both TP and TN, we measure *accuracy* (A_q) which is the degree of closeness of results to the preferable values where all dependent and independent components are correctly identified. The higher the *accuracy*, the closer the prediction outcomes are to the perfect results where both FP and FN are equal to zero. *Accuracy* [19] is defined as follows:

$$A_q = \frac{TP_q + TN_q}{TP_q + FP_q + FN_q + TN_q}$$

DOI: 10.1002/smr.1598



Legend: A high level architectural analysis reveals that ADEMPIERE is a highly complex Java system built on and dependent on the older Compiere core.

Figure 7. ADEMPIERE Architecture: High Level View

4.2. Case Study: ADEMPIERE

We scouted the open-source software landscape for a suitable open-source system to use as a case study for our analysis. After considering several candidates, we eventually settled on ADEMPIERE, an Enterprise Resource Planning (ERP) software package. The qualities that persuaded us to choose ADEMPIERE for our case study are:

Well defined business domain: An ERP system integrates internal and external management information across an entire organisation, embracing accounting, manufacturing, sales and service, *etc.* Such a system manifests a strict separation between the expertise of the stakeholders and developers. This is the type of software which benefits mostly from domain-based coupling analysis.

Tiered architecture: The system manifests a clear separation between the different architectural tiers. The system has a rich set of UI components and four distinct front-ends from which the user can choose including a Java GUI and three web interfaces. The system heavily uses relational database management systems (*e.g.,* PostgreSQL and Oracle) for data storage as well as for storing business logic.

Evolving and active system: The ADEMPIERE project traces its evolution back more than a decade. Created in September 2006 as a fork of the Compiere open-source ERP, itself founded in 1999, ADEMPIERE soon reached the top five of the *SourceForge.net* enterprise software rankings. At the time of this publication, it is the first system among that top five. This is a measure of both the size of its developer community and its impact on the ERP software market.

Large-scale and complex design: The system represents cutting edge open-source software technology. It is a multi-language system that aggregates more than 2 million lines of code. The core part is written in Java and contains more than $3,531$ classes with more than half a million lines of code[§].

Figure 7 presents a high-level architectural view of the Java core of ADEMPIERE as obtained with the architecture recovery tool Softwarenaut [20]. The view is obtained by aggregating the direct call and inheritance relationships in the system up along the package hierarchy. The area

[§]Measured based on the Java code in the SVN repository at: `https://adempiere.svn.sourceforge.net/svnroot/adempiere/tags/trunk_last/`

DOI: 10.1002/smr.1598

of every visible module is proportional to its number of lines of code. Every visible dependency is directed and has its width proportional to the number of abstracted low-level dependencies. Every module is represented similar to a treemap, with the sizes of the contained classes and modules proportional to their size in lines of code.

Active developers and users community: The system has a very active associated community: often the mailing list has more than 800 messages per month, and the *SourceForge.net* page shows that ADEMPIERE is downloaded more than $15,000$ times per month. The system is used by a large number of companies around the world.

For all these reasons, we deem ADEMPIERE to be relevant and representative for enterprise systems and for the state of the art in open-source software at the moment of writing this article, and appropriate for our analysis.

4.3. Macro Evaluation

In order to evaluate the results for all UICs in ADEMPIERE, we take the mean value of measurements of all queries as

$$f_M = \frac{1}{n} \sum_{i=1}^{n} f_{q_i}$$

where f is one of these measurement functions: TP, TN, FP, FN, R, P, F_1 or A.

4.4. Likelihood

One application of domain-based coupling might be notifying software maintainers of possible dependent components when they browse a list of UICs. To assess the usefulness of such notifications, we measure the *likelihood* (L) whether at least one of the top three, five or ten returned results have architectural dependencies. More formally if $AN_{c,n}$ shows the top n results for a component c, then

$$L_n = \frac{|\{c|c \in C, AN_{c,n} \cap E_c \neq \emptyset\}|}{|\{c|c \in C\}|}$$

The *likelihood* function enables us to distinguish between the topmost results and the entire returned result set.

4.5. Results: Searching For Source Code Dependencies

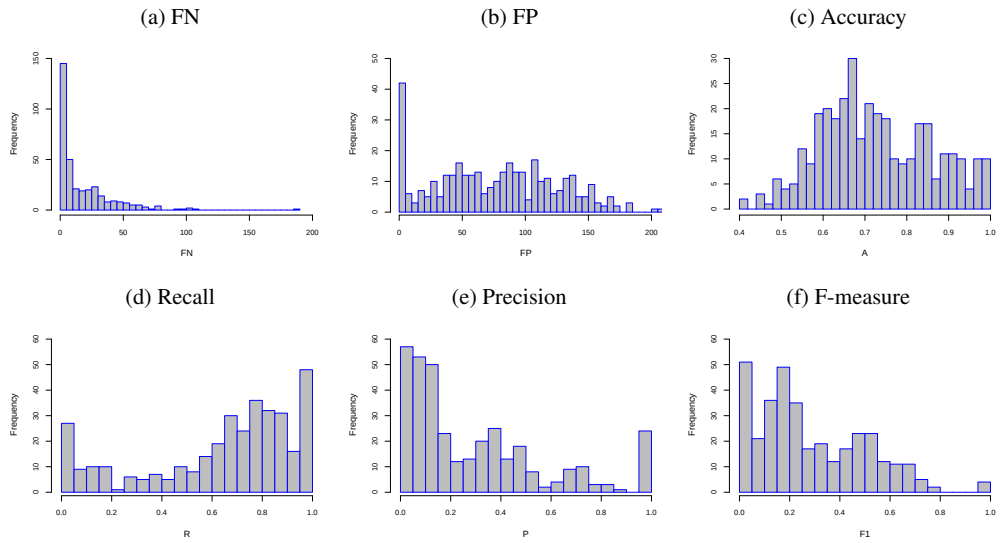
In this section, we investigate the first research question: *How accurately can we predict dependencies between UICs at the source code level?* ADEMPIERE contains 348 UICs. The source code analysis revealed $16,450$ indirect dependencies and no direct dependencies among classes behind these UICs. For any given UIC, we queried the connected UICs by source code dependencies and compared the results with the domain-based coupling graph.

	ALL	FILTERED		ALL	FILTERED
TP_M	31±36.84	31±36.84	A_M	0.73±0.13	0.73±0.13
FN_M	18±23.05	18±23.05	F_{1M}	0.29±0.22	0.29±0.21
FP_M	77±48.75	77±48.75	FB	0.93	0.96
TN_M	222±69.42	222±69.42	L_3	0.72	0.74
P_M	0.30±0.28	0.30±0.28	L_5	0.77	0.80
R_M	0.63±0.30	0.63±0.30	L_{10}	0.83	0.86

Legend: ALL: complete queries for all ADEMPIERE's UICs. FILTERED: only queries with $|AN| > 0$.

Table I. Source Code Dependencies: Evaluation Results

Prepared using smrauth.cls
DOI: 10.1002/smr.1598



Legend: x-axes are the measures and the y-axises are the frequency of the values, *i.e.,* subfigure (a)FN shows that there are 145 queries with less $FN < 5$.

Figure 8. Source Code Dependencies: Histogram Of The Queries' Outcome.

Figure 8 shows the histogram of the queries' outcomes. As presented in Figure 8a, the larger majority of the query results contain little false negatives, for example, 109 queries have less than 3 false negatives. A comparison between the histograms of recall and precision shows that the domain-based coupling makes conservative predictions, *i.e.,* for most of the queries there is a trade off between recall and precision in favour of recall. Moreover, Figure 8c shows the high accuracy of the results for most queries. Such predictions are particularly useful for change impact analysis where high recall provides confidence about the prediction of the scope of change propagation. The feedback is 0.93 which shows that for 93% of queries domain-based coupling returned at least one correct answer.

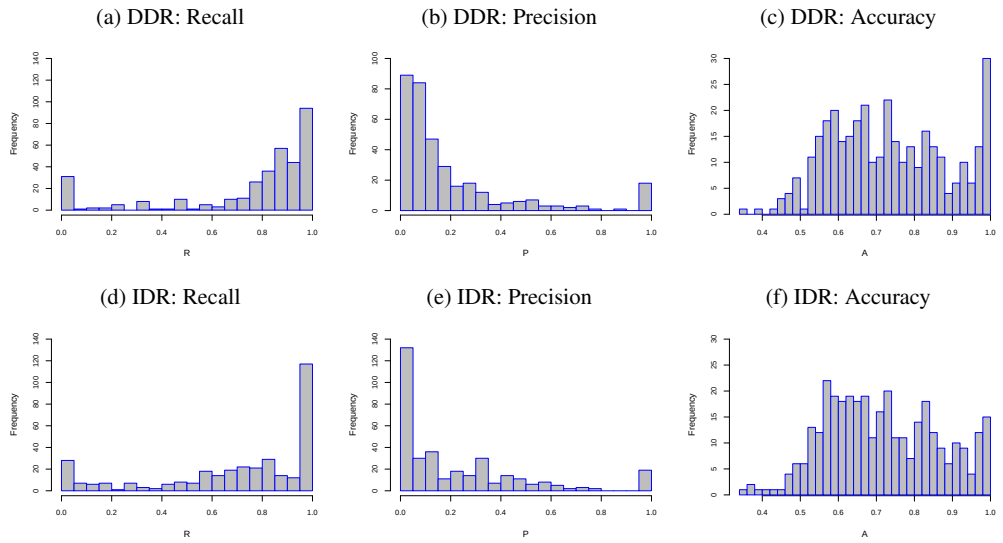
For example, *Financial Report* is a UIC in ADEMPIERE with source code dependencies to 8 other UICs. The query for this UIC returned 60 UICs including 7 true positive and 53 false positive results, leading to the precision of 0.12. Given that there are 348 UICs in ADEMPIERE, the accuracy of the results is 0.84. This strong accuracy indicates that the query results allow a software maintainer to focus on limited number of UICs (60 UICs rather than 348 UICs).

On average for a given UIC, 31 connected UICs by source code dependencies are identified correctly while 18 UICs with source code dependencies are incorrectly described as independent components, and 77 independent UICs are falsely identified to have source code dependencies. These results lead to average recall equal to 0.63, average precision equal to 0.30 and average F-measure of 0.29.

The *accuracy* of the prediction is equal to 0.73, implying that for more than 7 out of 10 UICs, our prediction method correctly identified whether two UICs are dependent or independent at the source code level. The likelihood of discovering source code dependencies in the top three coupled UICs is 72%, and it will increases to 83% for the top ten UICs.

In any given system, it is expected to find some independent UICs, *i.e.,* $E = \emptyset$. The queries for these UICs might distort the average results. In ADEMPIERE, we identified 16 UICs that have no source code dependencies to any other UICs. We filtered out these queries and measure the average results for the rest of the queries. The comparison between the average results for all queries and the filtered queries (Table I) shows a minor decrease in recall and almost no change in average precision, F-measure and accuracy. Moreover, the feedback only increases from 0.93 to 0.96, and the likelihood of finding source code dependencies between top three, five and ten results slightly increases by

DOI: 10.1002/smr.1598



Legend: DDR: Direct Database Relationships, IDR: Indirect Database Relationships.

Figure 9. Database Dependencies: Histogram Of The Queries' Outcome.

Direct Relationships	ALL	FILTERED		ALL	FILTERED
TP_M	19±25.96	19±26.19	A_M	0.74±0.15	0.74±0.15
FN_M	4± 9.61	4± 9.76	F_{1M}	0.23±0.21	0.23±0.20
FP_M	87±53.18	88±52.45	FB	0.91	0.96
TN_M	238±66.79	236±66.30	L_3	0.58	0.59
P_M	0.20±0.25	0.2±0.24	L_5	0.66	0.68
R_M	0.77±0.30	0.8±0.30	L_{10}	0.75	0.77
Indirect Relationships	ALL	FILTERED		ALL	FILTERED
TP_M	21±29.95	28±31.60	A_M	0.72±0.15	0.72±0.14
FN_M	13±18.81	17±19.97	F_{1M}	0.22±0.23	0.28±0.20
FP_M	85±53.51	81±51.12	FB	0.93	0.95
TN_M	229±68.46	222±69.82	L_3	0.51	0.66
P_M	0.22±0.27	0.3±0.26	L_5	0.56	0.73
R_M	0.71±0.32	0.6±0.31	L_{10}	0.62	0.81

Legend: ALL: complete queries for all ADEMPIERE's UICs. FILTERED: only queries with $|AN| > 0$.

Table II. Database Relationships: Evaluation Results

2% or 3%. Therefore, we conclude that the queries with $E = \emptyset$ have a little impact on the overall prediction results.

Summary: *On average* 63% *of UICs connected by source code dependencies has been identified correctly, while for* 83% *of queries the top ten results contains one or more source code dependencies.*

4.6. Results: Searching For Database Relationships

In this section, we address the second research question: *How accurately can we predict relationships between UICs at the database layer?* The database analysis of ADEMPIERE showed that there are $7,986$ direct and $11,894$ indirect relationships among data tables behind UICs. We queried these relationships using the domain-based coupling, and the results are presented in Table II. The feedback of these queries is 0.91 and 0.93 for direct and indirect relationships respectively.

DOI: 10.1002/smr.1598

TP_M	31±36.91	P_M	0.31±0.29	FB	0.93
FN_M	19±23.91	R_M	0.63±0.30	L_3	0.72
FP_M	75±49.53	F_{1M}	0.30±0.22	L_5	0.78
TN_M	223±70.19	A_M	0.73±0.13	L_{10}	0.84

Table III. Architectural Dependencies: Evaluation Results

On average for a given UIC, 19 directly related UICs and 21 indirectly related UICs are identified correctly. The results show only 4 false negatives for direct relationships that is more than three times lower than the 13 false negatives identified for indirect relationships. However, the number of false positives is similar: 87 and 85 for direct and indirect relationships respectively.

Comparing the results between direct and indirect relationships shows that for direct relationships the recall is slightly higher (0.77 vs. 0.71), while the precision is slightly lower (0.20 vs. 0.22). The accuracy values for both relationship types are higher than 0.7, which means that the relationship state of 7 out of 10 UIC pairs is identified correctly. In addition, validating the topmost results shows that the likelihood of database relationships in the top three results is 58% for direct and 51% for indirect relationships. Also the likelihood of indirect relationships increases to 75% for the top ten results.

For example, *Financial Report* (a UIC in ADEMPIERE) has 9 direct database relationships to other UICs. Our query returns 60 UICs which includes 8 UICs with database dependencies to Financial Report. The recall computed from these results equals to 0.89 and precision equals to 0.13. Although the precision is low, but the query results includes 287 true negatives that leads to the accuracy of 0.85. Such a high accuracy can assist software maintainers to exclude independent components from change impact analysis.

Figure 9 shows the histogram of precision, recall and accuracy of queries for direct and indirect database dependencies. As it is illustrated in Figures 9a and 9b there are noticeable number of queries with recall equals to one and precision close to zero, *i.e.,* their results contain no false negatives and some false positives. Qualitative analysis of the results shows that the recall of 84 queries for direct database relationships equals to one from which 13 queries have no expected answer, *i.e.,* $E = \emptyset$. Figures 9d and 9e show a similar but accentuated pattern. The recall of 116 queries for indirect database relationships equals to one from which 86 queries have no expected answers. Such queries might distort the results. We measured their impact on the average results by filtering them out from result set as presented in Table II. The comparison shows minor increases in the average true positives and false negatives, and minor decreases in false positives and true negatives. The impact of these changes on precision and recall for direct dependencies is not noticeable; however, the precision and the F-measure of the indirect dependencies slightly increases, while their recall slightly decreases.

Summary: *On average more than* 71% *of database relationships can be derived from domain information, and the likelihood of finding database dependencies among the top ten results is up to* 75%.

4.7. Results: Searching For Architectural Dependencies

In this section, we address the third research question: *How accurately can we predict architectural dependencies between UICs?* The analysis of the source code and the database of ADEMPIERE shows 17,279 architectural dependencies (Definition 8). We evaluated how accurately a domain expert can predict whether there is at least one architectural dependency between any given pair of UICs. Figure 10 shows the precision, recall, F-measure and accuracy of the query results based on the number of UICs in the returned answer by the query, *i.e.,* x-axis shows the $|AN| = TP + FP$.

As it is illustrated in Figures 10a, the recall significantly increases proportionally to the increment of the result size, however, this behaviour is different for the precision (Figure 10b). Therefore, the average F-measure (Figure 10c) slightly increases for queries with higher number of results. On the contrary, Figure 10d shows that there is a clear negative relationship between the size of the

DOI: 10.1002/smr.1598

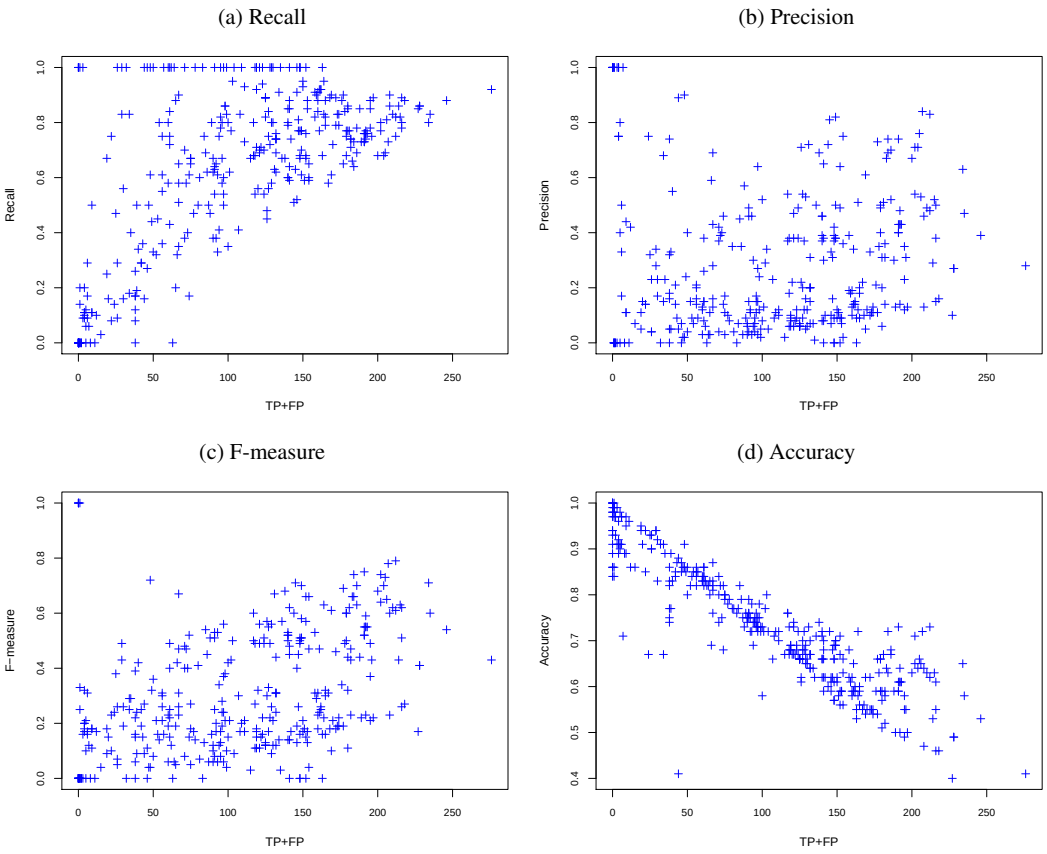


Figure 10. Architectural Dependencies: Scatter Chart Of Evaluation Metrics Based On The Size Of The Result Set.

results returned by the queries and the accuracy of the results themselves. We can conclude that in ADEMPIERE, UICs with less domain-based coupling to other UICs are more likely to be independent components at the architectural level.

The feedback of the queries is 0.93, *i.e.,* for 93% of queries the domain-based coupling returned at least one correct answer. As presented in Table III, on average for a given UIC, 31 dependent UICs, and 223 independent UICs are identified correctly using domain information. However, 19 dependent and 75 independent UICs are incorrectly placed in the opposite dependency state. These results lead to an average recall of 0.63 and precision of 0.31. The average accuracy of the predictions is 0.73, that shows for 7 in 10 UIC pairs their dependency state is identified correctly. In addition, the likelihood of discovering an architecturally dependent UIC pair in the top three results is 72%. This likelihood will increase to 84% for the top ten results.

Summary: *On average* 63% *of architecturally dependent UICs are discovered using domain information, and the likelihood of discovering a correct architectural dependency in the top ten predictions is* 84%.

4.8. Improving Precision

The prediction results for architectural dependencies (Table III) show that the average precision is 0.31. In order to improve the *precision*, we utilised the expectation maximisation technique (Section 2.5) to filter out weakly coupled pairs, with the assumption that UICs with strong domain-based coupling are more likely to have architectural dependencies.

DOI: 10.1002/smr.1598

	R_M	P_M	A_M
Source Code Dependencies	0.29	0.68	0.88
Direct Database Relationships	0.40	0.57	0.89
Indirect Database Relationships	0.27	0.61	0.93
Architectural Dependencies	0.23	0.70	0.87

Table IV. Prediction Results Using EM Clustering

Table IV shows the improved results. The mean *precision* for architectural dependencies is increased from 0.31 to 0.7 and the mean *accuracy* is increased from 0.73 to 0.87. However, these improvements are achieved at the expense of the reduction in *recall*. While the value of *precision* is more than doubled, the value of *recall* decreased almost three times (from 0.64 to 0.23). This implies that there are a number of architectural dependencies between UICs that have no strong coupling at the domain level.

Summary: *By using expectation maximisation technique, precision can be improved up to* 0.7. *However, it is a trade-off between precision and recall.*

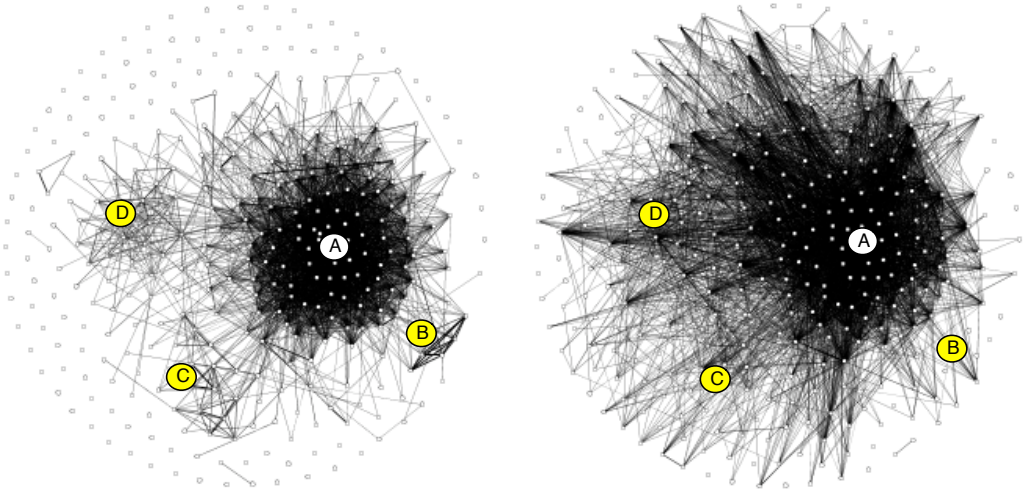
4.9. Results: Visual Comparison

In this section we visually compare the results obtained with the domain-based coupling against the actual architectural dependencies between software elements. The visualization in Figure 11 provides the reader with a graphical answer to the third research question: How accurately can we predict architectural dependencies between UICs?

The domain-based coupling graph (Figure 11a) is visualized using Fruchterman and Reingold's [21] force-based graph layout in three steps: first, the graph is created based on Definition 3; second, the exception maximisation (EM) technique (Section 2.5) is applied; third, the derived graph is visualised by the force-based layout algorithm.

(a) Domain-based coupling graph — EM applied

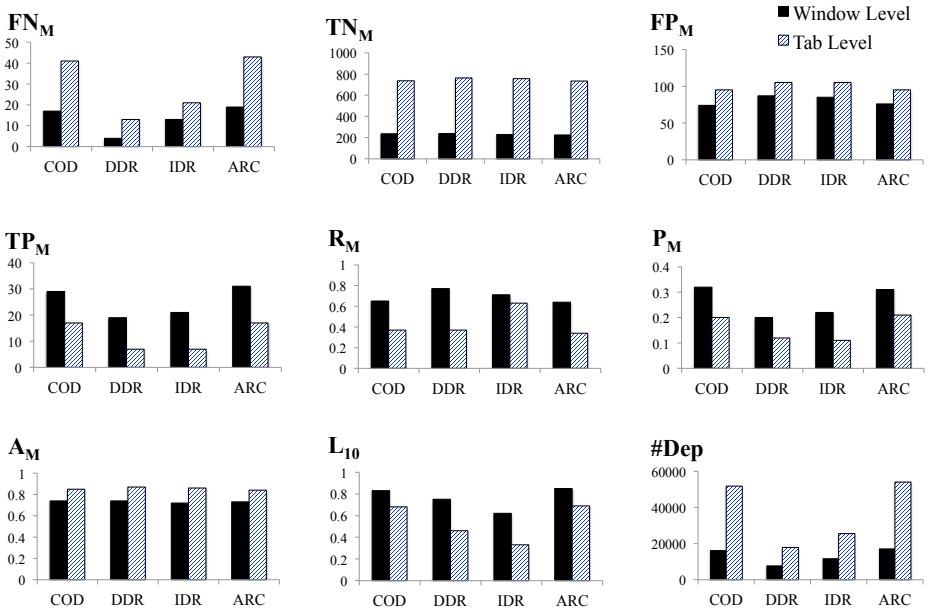
(b) Architectural dependency graph



Legend: Nodes are the UICs of ADempiere in both graphs. Left: Edges are domain-based coupling (Definition 3) which are selected by Expectation Maximization (Section 2.5). Right: Edges are architectural dependencies (Definition 8). Tags (A, B, C and D) are concentration areas.

Figure 11. Domain-Based Coupling vs Architectural Dependencies

DOI: 10.1002/smr.1598



Legend: COD: Source code dependencies, DDR: Direct database relationships, IDR: Indirect database relationships, ARC: Architectural dependencies, #Dep.: Number of dependencies.

Figure 12. Impact of Granularity on Prediction Results

In order to compare the domain-based coupling graph with the architectural dependencies, the edges from Figure 11a are replaced with the architectural dependencies without changing the location of nodes. The resulting graph (Figure 11b) illustrates the distribution of the architectural dependencies in compare to the domain-based coupling.

The comparison between Figure 11a and Figure 11b shows that the most populated cluster (tagged by A) in the domain-based coupling graph has the biggest number of architectural dependencies. However, the number of architectural dependencies decreases in the clusters with poor domain-based coupling (B, C and D). In addition, there are a number of architectural dependencies where there is no domain-based coupling, illustrating that not all dependencies can be derived from the domain-based coupling graph.

4.10. Results: Impact of Granularity

In this section, we address the fourth research question: *What is the impact of granularity of UICs on the prediction results?*

In Section 2.4, we described two granularity levels for UICs in ADEMPIERE. The coarse-grained UICs are *windows*, and each window is composed of fined-grained UICs called *tabs*. We evaluate the impact of granularity by repeating the queries from Section 4.5, Section 4.6 and Section 4.7 for the fine-grained UICs. ADEMPIERE is composed of 889 tabs, which is more than twice the number of the 348 windows. As such, the number of architectural dependencies between tabs is $54,030$ that is more than three times higher than the $17,279$ architectural dependencies between windows.

This increase in the number of dependencies has a notable impact on the prediction results. Figure 12 shows that the true negatives for the fine-grained UICs are improved by more than 200% in comparison to coarse-grained UICs. Consequently, the overall accuracy of the queries is increased by 14% to 20%. However, the average number of false negatives for code dependencies and database relationships for the fined-grained UICs is more than twice that for coarse-grained UICs. Moreover, the average number of false positives is increased by 20% to 28%. As a result of the increase in

Prepared using smrauth.cls
DOI: 10.1002/smr.1598

the false positives and false negatives, the average recall and precision is reduced by 11% to 50%. Furthermore, the likelihood of finding dependencies between the top 10 coupled UICs is reduced by 18% to 46%.

These results suggest that the proposed approach provides a better outcome for coarse-grained UICs. The overall accuracy increases slightly for the fined-grained UICs, but the noticeable decrease in the precision and recall might discourage software maintainers from using this method.

Summary: *Domain-based coupling provides a more precise prediction of dependencies between coarse-grained UICs than fine-grained UICs.*

4.11. Discussion

The reverse engineering of ADEMPIERE revealed $16,450$ source code dependencies, $7,986$ direct and $11,894$ indirect database relationships among ADEMPIERE UICs. We used the FAMIX meta model (Section 3.4) and the Definition 8 to identify how UICs are architecturally connected via the database and source code dependencies. The results show $17,279$ architectural dependencies between ADEMPIERE UICs.

In this evaluation, we queried these dependencies using domain-based coupling. The results shows that for more than 90% of queries the domain-based coupling returns at least one correct answer. The average recall for these queries is more than 0.6 for both source code dependencies and database relationships, while the precision and F-measure is lower than 0.4. Although the precision is not strong, but the average accuracy of the queries is higher than 0.7. The accuracy reflects both true positives and true negatives, and where there are many components in a system the number of true negatives is important for maintenance activities and particularly change propagation analysis.

For example, a domain expert need to estimate the impact of a change to a *Financial Report*, a UIC in ADEMPIERE. Domain-based coupling graph of ADEMPIERE shows 60 coupled UICs to Financial Report, and evaluating the source code and database shows that this result includes one false negative, 53 false positives, 7 true positives and 286 true negatives. From these results, we compute the accuracy of the query as 0.84. This high accuracy shows that the outcome of this query enables the domain expert to focus on a significantly reduced search space for change propagation (60 rather than 348 UICs).

Comparison between the size of the queries' results and the precision, recall, F-measure and accuracy shows that queries with larger outcome have higher recall and lower accuracy. However, the relationship between the size of the queries' results and the precision and F-measure is less significant. In addition, we have evaluated the impact of granularity of UCs, and the results show that domain-based coupling provides a more precise prediction of dependencies between coarse-grained UICs than fine-grained UICs.

One of the factors that affect the average results is the number of independent UICs. The expected answers for these queries are empty sets, which leads to a recall of 1, and any false positive for these queries leads to precision of zero.

We performed a qualitative analysis and filtered out these queries from our result set. The comparison between the results showed a small change in average precision, recall and accuracy. However, filtering out these queries improved the feedback (FB) and increased the likelihood of finding dependencies in the top three, five and ten results.

The probability of finding architectural dependencies using domain-based coupling is a trade off between precision and recall in favour of recall. However, in some case it is preferable to achieve a more precise result set. For example, if someone aims to develop a tool based on this method, too many false positives can discourage the user. In Section 4.8, we demonstrated how an unsupervised clustering method can be used to automatically increase the precision to 0.7 at the cost of the reduction in recall.

Finally, we demonstrated how domain-based coupling could be used to inform software maintainers while they browse software UICs. The results show the likelihood of discovering architectural dependencies among the top ten coupled UICs is 85%. Given that these results are obtained without looking at the source code or the database, they are quite promising. On the other hand in its current form, domain-based coupling analysis cannot completely replace source code analysis.

DOI: 10.1002/smr.1598

5. APPLICABILITY

In this paper, we introduced a novel approach to dependency analysis based on the analysis of domain-based coupling between UICs. Now we answer the following questions about the applicability of this approach:

- *What are the requirements for implementing this approach?* This approach requires access to the system domain knowledge including access to the information about the UICs, their domain functions and their associated domain variables. For most enterprise systems, domain experts are the best source of information about the software domain functionalities; in addition, the information about UICs and domain variables can be derived from the system user manuals or software help documents. Therefore, this approach requires access to the domain experts or alternative sources of information about UICs and their domain functions.
- *What kinds of software can benefit the most from the proposed approach*?
  Lehman [22] classified software into three types: *S-Type software* can be validated relative to a formal specification, and includes systems such as compilers. *E-Type software*, such as enterprise systems, is designed to mechanize a human or societal activity. *P-Type software* is a intermediate between S-Type and E-Type systems, such as a chess game, where users are concerned with the execution results rather than validating the implementation. The S-Type and P-Type system often have limited user interfaces, and most cases they operate based on a model which is not visible to the end user. The proposed approach in this thesis is suitable for systems with many UICs which enable end users to manage information based on domain driven business rules and workflows. Therefor, this work does not consider S-Type and P-Type software systems, and it focuses only on E-Type software. Most E-Type systems interact with human users through a number of UICs, and they are mostly data driven systems that collect, manage and report domain information. These systems take the most benefit from the domain-based coupling analysis.
  In a more detailed classification, Pressman [23] classified computer software into seven categories: system, application, engineering/scientific, embedded, product-line, artificial intelligence, and web applications. The domain-based approach is applicable to subsets of application software, product-line software and web applications, which are data-driven and provide their functionality through a number of user interface components.
  Domain-based coupling analysis is not applicable to software systems that their functionality is not visible to domain users, such as system software or embedded software. Also, it may not be suitable where systems are not data-driven or have few user interface components, such as engineering/scientific or artificial intelligence systems.
- *What kinds of software changes and maintenance activities can benefit the most from the proposed approach?* We envisage that the main application of the proposed approach will be estimating the change propagation prior to maintenance activities such as bug fixes and software enhancements. Lientz and Swanson [24] classified software changes as perfective, adaptive, corrective and preventative. Preventative changes are typically initiated by programmers/developers or software engineers who are concerned with the non-functional properties of the system, such as the maintainability of the source code. Such changes might be difficult to map to domain functions and UICs; therefore, the proposed approach would not be suitable for this kind of changes. However, perfective, adaptive and corrective changes are typically performed in response to a request from the system users or in response to changes in the software environment. Such software changes are often easy to map to UICs; therefore, domain experts can use the proposed approach to analysed the coupled UICs and estimate the change propagation.

In summary, the domain-based coupling analysis is applicable to the most enterprise systems and data driven software packages which provide most of their functions through the UICs. For such systems, domain experts might use the domain-based coupling to predict the dependencies between UICs, and estimate the impact of perfective, adaptive and corrective software changes.

DOI: 10.1002/smr.1598

6. THREATS TO VALIDITY

In this section, we discuss the threats to validity of our findings, and how we addressed them.

Threats to *external validity* are concerned with generalisation of our findings. Although we performed our evaluation on a large-scale enterprise system representative of the state of the art enterprise systems developed in Java, we are aware that more studies are required to be able to generalise our findings.

Threats to *construct validity* are concerned with the quality of the data we analysed, and the degree of manual analysis that was involved. The domain information typically is provided by the domain experts using a manual data collection process. To minimise the risk of human error, we extracted the relationship between domain variables and UICs from user manuals and help documents. In ADEMPIERE, this information is stored in the database. We only used manual inputs from domain experts to confirm this information and kept the manual additions and alterations to a minimum.

One other factor that could affect the validity of our results is the type of domain information in the case study. We limit the domain analysis to the domain variables visible on the UICs; however, there are other sources of domain information including user manuals and help files. It can be argued that different results might be achieved using alternative information; hence, further studies are required to evaluate various data sources and to identify the most suitable source for domain analysis.

7. RELATED WORK

The key applications of dependency analysis are change impact analysis, program comprehension, concept location and reverse engineering [25, 26, 27, 28]. Over the last two decades researchers have proposed different techniques to perform such analysis.

The earliest techniques rely on formal models of change propagation. LUQI [29] presented a graph model for software evolution based on indirect relationships between components. Rajlich [4] introduced a model for change propagation based on graph rewriting which requires an understanding of the dependencies between software elements. Arnold and Bohner [30] modelled change impact analysis as a cycle of revisions derived from relationships between software elements. Mirarab *et al.* [31] introduced a hybrid impact analysis method based on dependency information and co-change history.

More recent techniques work at the source code level. Source code analysis [6] is an established approach for tracing software dependencies [32, 33] or evaluating the evolution of code and design [34, 35]. One of the code analysis methods is *program slicing*, which has been exhaustively explored by many researchers and extended to many programming paradigms [36, 37, 38, 39]. Source code analysis is further enhanced using dynamic analysis [40, 41] to capture dependencies which might not be traceable from static relationships between software elements.

One direction in which impact analysis is extended is towards the analysis of entire software ecosystems [42]. In their work [43], Robbes *et al.* reported on an empirical study on the impact of API deprecations in a large open-source software ecosystem and conclude that tool support for impact estimation is needed also at the ecosystem level.

Further studies provided techniques based on software metrics. In an effort to quantify the dependency between objects, researchers and practitioners have defined metrics like Coupling Between Objects (CBO) or CBO' [44] which consider the inheritance between classes to measure the coupling among software elements. Other metrics like Response For Class (RFC) [45] and RFC_∞ [44] consider indirect relations among classes based on a level of indirection in the invocation chain of the class methods. A good overview of the structural coupling metrics is provided by Briand *et al.* [46]. Often these metrics attempt to provide feedback on the quality of a systems design. In our case although we provide a *coupling* metric between GUI components we do not aim at providing any feedback on the quality of the system but rather we use that metric to support the evolution of software.

The previously presented approaches rely on the source code being available. They also assume that the source code captures the relationships between the software elements. However, different

DOI: 10.1002/smr.1598

parts of the system change together even when the source code encodes no explicit dependencies between them (e.g. source code and configuration files). To detect the impact of a change when there are no source code relationships between the components involved in the change, one can use logical coupling [47, 48] or dynamic coupling [49, 50]. Several techniques [51, 52, 53, 54, 27] use an evolutionary approach which analyze multiple versions of a system by mining its software repositories. These techniques work under the assumption that the parts of the system that frequently change together will keep changing together in the future. Just as our approach, these approaches are less expensive, and require less technical expertise, than the ones based on data flow and source code analysis. On the other hand, unlike ours, they are not applicable where maintenance history is not accessible.

An alternative to studying the evolution of a system is to define the conceptual coupling metric based on the vocabulary of the different components included in a software system. Poshyvanyk *et al.* [55, 56, 57] and Gethers [58] identified and measured the relations between software entities in object-oriented software using topics included in the source code and latent semantic indexing. The difference between the domain-based coupling approach presented in this paper and the conceptual coupling approach is that the former is source code independent. In a recent study on ADEMPIERE, Gethers *et al.* [59] have demonstrated that domain-based coupling and conceptual coupling are orthogonal, and combining them leads to better prediction of database and source code dependencies with higher precision and recall as compared to its standalone constituents.

Gall *et al.* had shown that semantics metrics computed on design documents correlate well with semantic metrics computed on the source code, thus could be used as proxies for them [60]. Like domain-based coupling also this approach does not require source code analysis to compute coupling metrics. On the other hand, the approach of Gall *et al.* works on design specifications which can be outdated, most of the time not even available in first place. The domain-based coupling is computed starting from user interface components that are necessarily updated to the latest features offered by a system.

8. CONCLUSION AND FUTURE WORK

In this paper, we demonstrated how domain information could be used to predict architectural dependencies, and assist software maintainers in searching for connected components at the source code or the database layers. Our proposed approach for predicting dependencies promises independence from software implementation and simplicity and usability for non-technical domain experts. Hence, it can assist managers and consultants to take decisions about software changes without the support of the developers.

The proposed dependency analysis method is based on relationships between software domain information and user interface components (UIC), modelled as a weighted graph. We demonstrated how such a model can assist in predicting dependencies with a case study on a large-scale enterprise system, called ADEMPIERE. We derived architectural dependencies as a set of source code and database dependencies, and compared them with domain-based coupling between UICs. The results show that on average 65% of the source code and up to 77% of the database dependencies could be derived from the domain-based coupling. The accuracy of such predictions is on average more than 70%, implying that for 7 out of 10 component pairs their dependency state is identified correctly. The results promise that domain information might be used to predict the existence of architectural dependencies, and the accuracy of these predictions could support maintenance activities such as change impact analysis. However, at the current stage, this approach cannot replace source code analysis or database analysis.

A future area of investigation is assessing the impact of multiple domains on the results. ADEMPIERE contains various modules which provide functions of multiple domains like ERP, CRM and Asset Management. Distinguishing these domains and their domain-based coupling graphs might lead to better understanding of the relationships between domain-based coupling and architectural dependencies. Moreover, we envisage that domain-based coupling can be used in combination with other coupling metrics to predict change propagation. Domain-based coupling

DOI: 10.1002/smr.1598

is described at the abstract level of software domain. Such an abstract coupling can complement code-based conceptual coupling or history-based evolutionary coupling since each of these metrics describes a distinguished aspect of software systems.

Overall, the positive results of the described case study suggest that domain-based coupling can be considered to be complementary to source code analysis, and can assist software maintainers where existing code analysis tools are not applicable.

ACKNOWLEDGEMENT

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 – Sept. 2012). Part of Amir Aryani's contribution to this work has been funded by an Australian Postgraduate Award (APA). We would also like to thank Dr. Margaret Hamilton, Nicholas May and Jorge Ressia for their comments on this paper and their support on this project.

REFERENCES

1. Hassan AE, Holt RC. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Softw. Engg.* Sep 2006; **11**(3):335–367, doi:10.1007/s10664-006-9006-4.
2. Zhifeng Yu VR. Hidden dependencies in program comprehension and change propagation. *Proceedings of the 9th International Workshop on Program Comprehension*, IEEE Computer Society: Washington, DC, USA, 2001; 293–299, doi:10.1109/WPC.2001.921739.
3. Vanciu R, Rajlich V. Hidden dependencies in software systems. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010; 1 –10, doi:10.1109/ICSM.2010.5609657.
4. Rajlich V. A model for change propagation based on graph rewriting. *Proceedings of the International Conference on Software Maintenance*, ICSM '97, IEEE Computer Society: Washington, DC, USA, 1997; 84–91.
5. Hassan A, Holt R. Predicting change propagation in software systems. *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, IEEE Computer Society Press: Los Alamitos CA, 2004; 284–293, doi:10.1109/ICSM.2004.1357812.
6. Binkley D. Source code analysis: A road map. *2007 Future of Software Engineering*, FOSE '07, IEEE Computer Society: Washington, DC, USA, 2007; 104–119, doi:10.1109/FOSE.2007.27.
7. Lehman M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE* Sep 1980; **68**(9):1060–1076.
8. Cook S, Harrison R, Lehman MM, Wernick P. Evolution in software systems: foundations of the spe classification scheme: Research articles. *J. Softw. Maint. Evol.* Jan 2006; **18**(1):1–35, doi:10.1002/smr.v18:1.
9. Aryani A, Peake I, Hamilton M. Domain-based change propagation analysis: An enterprise system case study. *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010; 1 –9, doi:10.1109/ICSM.2010.5609743.
10. Aryani A, Perin F, Lungu M, Mahmood AN, Nierstrasz O. Can we predict dependencies using domain information? *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, 2011, doi:10.1109/WCRE.2011.17.
11. Aryani A, Peake ID, Hamilton M, Schmidt H, Winikoff M. Change propagation analysis using domain information. *Proceedings of the 2009 Australian Software Engineering Conference*, ASWEC '09, IEEE Computer Society: Washington, DC, USA, 2009; 34–43, doi:10.1109/ASWEC.2009.31.
12. Bastian M, Heymann S, Jacomy M. Gephi: An open source software for exploring and manipulating networks. *International AAAI Conference on Weblogs and Social Media*, 2009.
13. Mahmood A, Leckie C, Udaya P. An efficient clustering scheme to exploit hierarchical data in network traffic analysis. *Knowledge and Data Engineering, IEEE Transactions on* Jun 2008; **20**(6):752 –767, doi:10.1109/TKDE.2007.190725.
14. Dempster AP, Laird NM, Rubin DB. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)* 1977; **39**(1):1–38, doi:10.2307/2984875.
15. Nierstrasz O, Ducasse S, Gîrba T. The story of Moose: an agile reengineering environment. *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, ACM Press: New York, NY, USA, 2005; 1–10, doi:10.1145/1095430.1081707. Invited paper.
16. Tichelaar S, Ducasse S, Demeyer S, Nierstrasz O. A meta-model for language-independent refactoring. *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, IEEE Computer Society Press, 2000; 157–167, doi:10.1109/ISPSE.2000.913233.
17. Marinescu C, Jurca I. A meta-model for enterprise applications. *SYNASC '06: Proceedings of the Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, IEEE Computer Society: Washington, DC, USA, 2006; 187–194, doi:10.1109/SYNASC.2006.3.
18. Perin F. MooseJEE: A Moose extension to enable the assessment of JEAs. *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010) (Tool Demonstration)*, 2010, doi:10.1109/ICSM.2010.5609569.
19. Manning CD, Raghavan P, Schütze H. *Introduction to Information Retrieval*. Cambridge University Press, 2008.
20. Lungu M, Lanza M, Nierstrasz O. Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming (SCP)* 2012; :available online, to appear in printdoi:10.1016/j.scico.2012.04.007. URL `http://scg.unibe.ch/archive/papers/Lung12b.pdf`.

DOI: 10.1002/smr.1598

21. Fruchterman TMJ, Reingold EM. Graph drawing by force-directed placement. *Softw. Pract. Exper.* Nov 1991; **21**(11):1129–1164, doi:10.1002/spe.4380211102.
22. Lehman MM, Ramil JF. Rules and tools for software evolution planning and management. *Ann. Softw. Eng.* Nov 2001; **11**(1):15–44, doi:10.1023/A:1012535017876.
23. Pressman RS. *Software Engineering: A Practitioner's Approach*. Seventh edition edn., McGraw-Hill, 2010.
24. Lientz BP, Swanson EB. *Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Publishing Company: Reading, MA, USA, 1980.
25. Cleary B, Exton C. Assisting concept location in software comprehension. *Proceedings of 19th Annual Psychology of Programming Workshop (PPIG 07)*, Joensuu, Finland, 2007.
26. Tzerpos V, Holt R. Accd: an algorithm for comprehension-driven clustering. *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, 2000; 258 –267, doi:10.1109/WCRE.2000.891477.
27. Walker RJ, Holmes R, Hedgeland I, Kapur P, Smith A. A lightweight approach to technical risk estimation via probabilistic impact analysis. *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, ACM: New York, NY, USA, 2006; 98–104, doi:10.1145/1137983.1138008.
28. Marinescu C. Discovering the objectual meaning of foreign key constraints in enterprise applications. *Reverse Engineering, Working Conference on* 2007; **0**:100–109, doi:10.1109/WCRE.2007.20.
29. Luqi. A graph model for software evolution. *IEEE Trans. Softw. Eng.* Aug 1990; **16**(8):917–927, doi:10.1109/32.57627.
30. Bohner SA, Arnold R. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.
31. Mirarab S, Hassouna A, Tahvildari L. Using bayesian belief networks to predict change propagation in software systems. *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, 2007; 177 –188, doi:10.1109/ICPC.2007.41.
32. Harman M, Binkley D, Gallagher K, Gold N, Krinke J. Dependence clusters in source code. *ACM Trans. Program. Lang. Syst.* Nov 2009; **32**:1:1–1:33, doi:10.1145/1596527.1596528.
33. Cleve A, Henrard J, Hainaut JL. Data reverse engineering using system dependency graphs. *WCRE 06: Proceedings of the 13th Working Conference on Reverse Engineering*, 2006; 157 –166, doi:10.1109/WCRE.2006.22.
34. Hammad M, Collard M, Maletic J. Automatically identifying changes that impact code-to-design traceability. *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, 2009; 20 –29, doi:10.1109/ICPC.2009.5090024.
35. Lungu M, Lanza M. Exploring inter-module relationships in evolving software systems. *Proceedings of CSMR 2007 (11th European Conference on Software Maintenance and Reengineering)*, IEEE Computer Society Press: Los Alamitos CA, 2007; 91–100.
36. Binkley D, Harman M. A survey of empirical results on program slicing. *Advances in Computers* 2004; **62**:105 – 178.
37. Willmor D, Embury S, Shao J. Program slicing in the presence of database state. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004; 448 – 452, doi:10.1109/ICSM.2004.1357833.
38. Xu B, Qian J, Zhang X, Wu Z, Chen L. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes* Mar 2005; **30**(2):1–36, doi:10.1145/1050849.1050865.
39. Silva J. A vocabulary of program-slicing based techniques. *ACM Computing Surveys* 2011; .
40. Xiao C, Tzerpos V. Software clustering based on dynamic dependencies. *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, 2005; 124 – 133, doi:10.1109/CSMR.2005.49.
41. Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 2009; **35**(5):684–702, doi:10.1109/TSE.2009.28.
42. Lungu M. Reverse engineering software ecosystems. PhD Thesis, University of Lugano Nov 2009. URL `http://scg.unibe.ch/archive/papers/Lung09b.pdf`.
43. Robbes R, Lungu M, Roetlisberger D. How do developers react to api deprecation? the case of a smalltalk ecosystem. *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012; 56:1 – 56:11, doi:10.1145/2393596.2393662. URL `http://scg.unibe.ch/archive/papers/Rob12aAPIDeprecations.pdf`.
44. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* Jun 1994; **20**(6):476–493, doi:10.1109/32.295895.
45. Chidamber SR, Kemerer CF. Towards a metrics suite for object oriented design. *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, ACM: New York, NY, USA, 1991; 197–211, doi:10.1145/117954.117970.
46. Briand LC, Daly JW, Wüst JK. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 1999; **25**(1):91–121, doi:10.1109/32.748920.
47. Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. *26th International Conference on Software Engineering (ICSE 2004)*, IEEE Computer Society Press: Los Alamitos CA, 2004; 563–572.
48. Gall H, Jazayeri M, Krajewski J. Cvs release history data for detecting logical couplings. *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, IEEE Computer Society: Washington, DC, USA, 2003; 13–.
49. Arisholm E, Briand L, Foyen A. Dynamic coupling measurement for object-oriented software. *Software Engineering, IEEE Transactions on* Aug 2004; **30**(8):491–506, doi:10.1109/TSE.2004.41.
50. Hassoun Y, Johnson R, Counsell S. A dynamic runtime coupling metric for meta-level architectures. *Software Maintenance and Reengineering, European Conference on* 2004; **0**:339, doi:10.1109/CSMR.2004.1281436.
51. Ying A, Murphy G, Ng R, Chu-Carroll M. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on* Sep 2004; **30**(9):574 – 586, doi:10.1109/TSE.2004.52.
52. Hindle DGA, Jordan N. Visualizing the evolution of software using softchange. *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*, ACM Press: New York NY, 2004;

DOI: 10.1002/smr.1598

336–341.
53. D'Ambros M, Lanza M, Robbes R. On the relationship between change coupling and software defects. *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, IEEE Computer Society: Washington, DC, USA, 2009; 135–144, doi:10.1109/WCRE.2009.19.
54. Kagdi H, Maletic J, Sharif B. Mining software repositories for traceability links. *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, 2007; 145 –154, doi:10.1109/ICPC.2007.28.
55. Poshyvanyk D, Marcus A, Rajlich V, Gueheneuc YG, Antoniol G. Combining probabilistic ranking and latent semantic indexing for feature identification. *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC '06, IEEE Computer Society: Washington, DC, USA, 2006; 137–148, doi: 10.1109/ICPC.2006.17.
56. Poshyvanyk D, Marcus A. The conceptual coupling metrics for object-oriented systems. *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, IEEE Computer Society: Washington, DC, USA, 2006; 469–478, doi:10.1109/ICSM.2006.67.
57. Poshyvanyk D, Marcus A, Ferenc R, Gyimóthy T. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering* Feb 2009; **14**(1):5–32, doi:10.1007/s10664-008-9088-2.
58. Gethers M, Poshyvanyk D. Using relational topic models to capture coupling among classes in object-oriented software systems. *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, IEEE Computer Society: Washington, DC, USA, 2010; 1–10, doi:10.1109/ICSM.2010.5609687.
59. Malcom Gethers AA, Poshyvanyk D. Combining conceptual and domain-based couplings to detect database and code dependencies. *12th International Working Conference on Source Code Analysis and Manipulation*, IEEE Computer Society Press, 2012; 144–153, doi:10.1109/SCAM.2012.27.
60. Gall C, Lukins S, Etzkorn L, Gholston S, Farrington P, Utley D, Fortune J, Virani S. Semantic software metrics computed from natural language design specifications. *Software, IET* Feb 2008; **2**(1):17–26, doi:10.1049/iet-sen: 20070109.

DOI: 10.1002/smr.1598