STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXII**, Number 1, 2017
DOI: 10.24193/subbi.2017.1.07

IDENTIFYING HIDDEN DEPENDENCIES IN SOFTWARE SYSTEMS

ISTVÁN GERGELY CZIBULA, GABRIELA CZIBULA, DIANA-LUCIA MIHOLCA, AND ZSUZSANNA MARIAN

Abstract. The maintenance and evolution of software systems are highly impacted by activities such as bug fixing, adding new features or functionalities and updating existing ones. *Impact analysis* contributes to improving the maintenance activities by determining those parts from a software system which can be affected by changes to the system. There exist *hidden dependencies* in the software projects which cannot be found using common coupling measures and are due to the so called *indirect coupling*. In this paper we aim to provide a comprehensive review of existing methods for *hidden dependencies identification*, as well as to highlight the limitations of the existing state-of-the-art approaches. We also propose an *unsupervised learning* based computational model for the problem of *hidden dependencies identification* and give some incipient experimental results. The study performed in this paper supports our broader goal of developing *machine learning* methods for automatically detecting *hidden dependencies*.

1. Introduction

Maintenance activities such as bug fixes, updating existing features and adding new ones make up the majority of time and costs allocated to a software project. Each of these changes usually affects only parts of the system, and determining the affected components (classes, modules, methods etc.) is not a trivial problem. *Impact analysis* tries to identify, given a component of a software system, the other components that would be affected by changes to the former [7]. Existing methods for impact analysis usually consider only direct coupling between components, but there also exists *indirect coupling* [36], which creates *hidden dependencies*, that cannot be found using regular

Received by the editors: May 3, 2017.

2010 *Mathematics Subject Classification.* 68N30, 68T05, 62H30.

1998 *CR Categories and Descriptors.* K.6.3 [**Management of computing and information systems**]: Software Management – *Software maintenance*; I.2.6 [**Computing Methodologies**]: Artificial Intelligence – *Learning*; I.5.3 [**Computing Methodologies**]: Pattern Recognition – *Clustering*.

Key words and phrases. Impact analysis, hidden dependencies identification, machine learning, clustering.

coupling measures. Yet, not identifying them can have serious negative consequences [8].

Analyzing program dependencies has an essential role in program comprehension, change propagation, or impact analysis [22]. The software components need to be understood in the context in which they are defined and this context is expressed by the dependencies between the software components. It is fundamental for the software maintainers to discover the system's dependencies and make corresponding changes to ensure that change has been correctly spread out and the software remains stable [37]. Among the software component dependencies, *hidden dependencies* are relationships between two seemingly independent components and are produced by a data flow inside of a third software component [37].

The aim of this paper is to provide a systematic literature review on *hidden dependencies identification* (HDI), highlighting the difficulty of the problem as well as the limitations of the current state-of-the-art in this field. We are also proposing a new computational model based on *unsupervised learning* for the problem of *hidden dependencies identification*. With the broader goal of applying *machine learning* [23, 24] methods for detecting parts of a software system which are not directly coupled, we also describe the evaluation measures usually used for assessing the performance of methods for detecting hidden dependencies.

The remainder of the paper is organized as follows. The description of the HDI problem, together with an illustrative example, are given in Section 2. Section 3 presents the current state-of-the-art in *hidden dependencies identification*. Section 4 contains a discussion on the limitations of existing approaches, introduces our new *machine learning* perspective upon the problem and gives our incipient experimental results. We outline the conclusions of our paper and the directions to continue the research in Section 5.

2. Problem statement and importance

A special class of program dependencies, called *hidden dependencies* (HD) were introduced by Yu and Rajlich in [37]. The authors have also given examples of the software changes that these kinds of dependencies propagate in the code [37]. HDs are particular type of data flows [31] which have an important role in software maintenance and evolution. HDs propagate changes among the application classes of a software system and these changes are hard to detect. As shown in [31], *hidden dependencies* are found even in well designed software systems like JUnit, Drawlets, and Apache FtpServer. Thus, it is of crucial importance for software developers to detect and understand such dependencies.

A task of major importance for software developers is to understand HDs since it contributes to ease the software maintenance and evolution process. Among the software change activities that consider software dependencies we mention [31]: impact analysis [7, 27], change propagation [28], regression testing [32].

Data flows are considered to be the basis of *hidden dependencies* [31]. Since the process of analyzing the data flow in a software is not an easy task, it is very likely that software developers omit HDs rather than more explicit dependencies, introducing, in this way, bugs into the software [31].

The omission of HDs has a major impact particularly for critical computing systems. An example is a bug that was introduced during the evolution of the Minimum Safe Altitude Warning software system (MSAW) and which caused, in 1997, an aircraft crash at the Guam International Airport [9]. It has been shown that a missed HD between two software components which seemed independent has caused the bug in MSAW software: one component activated the alarm at 55 nautical miles and another component deactivated the alarm at 54 nautical miles [9].

The problem of identifying HDs is a very complex one. This is primarily because there is no exact definition for what a *hidden dependency* is.

Different methods existing in the literature were developed for detecting particular types of hidden dependencies. For example, Kagdi and Maletic considered in [17] that there is a HD between two software entities (methods or application classes) if the entities were changed at the same time in the past. Two application classes were considered by Gall et al. to be dependent [11] if they were changed by the same author and in the same time interval.

Beer et al. [5] have proposed a method for generating test data for problems involving complex linear dependencies between variables. The authors have suggested that software developers could specify restrictions on the values of variables in the source code and use them to generate the test cases. The dependencies that the authors called "complex dependencies" are able to capture semantic information that is hard to detect using traditional techniques for program analysis.

Jenkov defines in [16] a *hidden dependency* as a dependency which cannot be seen from a class's interface. Another example of a hidden dependency is the dependency on a static singleton, or static methods from within a method. One cannot observe from the interface if a class depends on static methods or static singletons [16]. These type of dependencies are hard to detect for software developers, they can be discovered only by inspecting the code [16].

2.1. **Examples of HDs.** As shown in Figure 1, in JUnit 4.4, there is a hidden dependency between the methods *getTestHeader()* of class *Failure* (Figure 2) and the method *getDescription()* of class *CompositeRunner* (Figure 3).

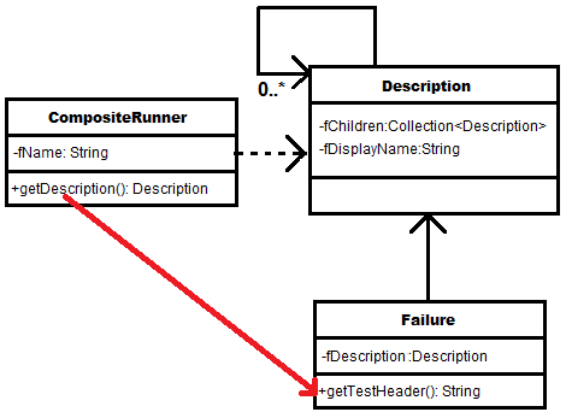

FIGURE 1. Hidden dependency in JUnit 4.4

```
package org.junit.runner.notification;

public class Failure{

    private final Description fDescription;
    ...

    /** @return a user-understandable label for the test*/
    public String getTestHeader() {
        return fDescription.getDisplayName();
    }
    ...
}
```

FIGURE 2. Failure.java

A *Description* object (see Figure 4) describes a test case or a test suite which is to be run or has been run. After execution, in case of failure, the name is printed after being decoded by the method *getTestHeader()* of the class *Failure*.

The method *getDescription()* creates a *Description* object and encodes *fName* in it. So, the method *getTestHeader()* will return it as a user-understandable label for the failed test. The methods share the *test case/suite* concept.

A justification considering pre- and postconditions, for the exemplified hidden dependency, is provided by Vanciu and Rajlich in [31].

3. LITERATURE REVIEW

In this section we present a literature review on the problem of *hidden dependencies identification* (HDI). Despite the importance of finding hidden

```
public class CompositeRunner extends Runner implements Filterable, Sortable {
    private final List<Runner> fRunners= new ArrayList<Runner>();
    private final String fName;

    public CompositeRunner(String name) {
        fName= name;
    }

    public void add(Runner runner) {
        fRunners.add(runner);
    }

    @Override
    public Description getDescription() {
        Description spec = Description.createSuiteDescription(fName);
        for (Runner runner : fRunners)
            spec.addChild(runner.getDescription());
        return spec;
    }
    ...
}
```

FIGURE 3. CompositeRunner.java

```
public class Description implements Serializable {

    private final Collection<Description> fChildren = new ConcurrentLinkedQueue<Description>();
    private final String fDisplayName;
    ...

    /** Create a <code>Description</code> named <code>name</code>.*/
    public static Description createSuiteDescription(String name, Annotation... annotations) {
        return new Description(name, annotations);
    }

    private Description(final String displayName, Annotation... annotations) {
        fDisplayName= displayName;
        fAnnotations= annotations;
    }

    /** @return a user-understandable label*/
    public String getDisplayName() {
        return fDisplayName;
    }

    /** Add <code>Description</code> as a child of the receiver.*/
    public void addChild(Description description) {
        fChildren.add(description);
    }
    ...
}
```

FIGURE 4. Description.java

dependencies, the approaches existing in the literature for this problem have moderate precision and recall values.

There are approaches which use previous versions of the software system and try to identify those classes which were changed together with respect to the same bug report [12]. Gall et al. have introduced in [12] an approach, called

CAESAR, that uses information about previous versions of a system to discover logical dependencies and change patterns among modules. The proposed method is experimentally evaluated on 20 releases of a large Telecommunications Switching System. Information such as version numbers of programs, modules and subsystems together with change reports are used for identifying common change patterns of software modules. CAESAR determines hidden dependencies which are not obvious in the source code, like modules that should be restructured. Instead of using the lines of code for the previous versions of the software, the authors use structural information about programs, modules and subsystems, together with change reports for the releases and their version numbers. The method proposed in [12] has been proved to be capable to identify bugs which were fixed in one version of the system but have appeared again, in other parts of the software, in later versions.

One of the early works is [37], where Yu and Rajlich have transformed System Dependence Graphs into Abstract System Dependence Graphs to determine which class pairs have hidden dependencies. The paper discusses how hidden dependencies impact the process of change propagation and also discusses an algorithm that indicates the possible presence of hidden dependencies. Hidden dependencies are considered to be design faults which contradict the rule "if a class A is unaware of the existence of class B, it is also unconcerned about any change to B". More exactly, a dependence between Class A and B is a hidden dependence if: (1) class A and B are not neighbors in the ASDG, i.e there is no direct dependence between A and B; and (2) there is a third class C, which is dependent on both classes, and there is data flow inside the class C that occurs between instance of class A and instance of class B. A simple algorithm for determining hidden dependencies is introduced and a JAVA example consisting of three classes collaborating to manage a session is considered.

In 2004, Hassan and Holt [14] have studied change propagation in software development. They have proposed several heuristics to predict change propagation by suggesting software entities that should be modified in accordance to the changes an entity has suffered. The heuristics have been empirically evaluated using historical data related to several open source projects. It has been experimentally shown that co-change data can be used to develop models for assisting software developers during change propagation process.

Orso et. al [25] have performed an empirical comparison of two existing dynamic impact analysis algorithms. Both algorithms use static analysis on the call-graph of the system, but they also use traces from the execution of the system to be analyzed. The first algorithm, CoverageImpact, constructs, for each execution, a vector with as many elements as methods in the system to be analyzed, and simply sets the value 1 for each executed method in this

vector. These vectors are used to determine the list of methods that were executed together with the method(s) that will be changed. This list is filtered using a static forward slice starting from each method to be changed. The second algorithm, PathImpact, constructs a so-called whole-path DAG (Directed Acyclic Graph) from the execution traces and this DAG is traversed, starting from the point that denotes the method to be changed, to determine which methods are impacted by the change. The authors have performed experiments using several versions of three Java systems to compare the precision, time and space cost of these two algorithms. The results showed that PathImpact is more precise (it returns a shorter list of methods affected by the changes to be performed in the system), but this precision comes at a significant cost of space (the whole-path DAGs need a lot more space to be stored than the binary vectors) and time. In [4], the same authors have introduced an approach that combines the precision of the PathImpact with the speed and small space overhead of the CoverageImpact method. In order to achieve this, they introduce the Execute After relation, defined for two entities, which is true if the first entity is executed after the second one. An entity can be impacted by a change to another entity only if this relation is true for them. The authors also propose a simple and fast algorithm to compute this relationship for every pair of entities and this can be done by keeping in memory only two vectors having as many elements as entities in the systems. Comparing the performance of this new algorithm to PathImpact, they conclude that it is as precise as PathImpact, but it is only slightly slower than CoverageImpact.

There are many different metrics to measure coupling between components of a software system, but most of these metrics measure direct coupling (according to [34], in a description containing almost 30 coupling metrics, only two mention indirect coupling). Indirect coupling is often considered to be simply the transitive closure of entities in direct coupling, but in many cases such transitive closures contain most of the entities from the system. Since indirect coupling can affect the maintainability of a software system as well, the authors of [34] have proposed an algorithm to detect one type of indirect coupling, which they call use-def coupling. By a simple example, they show that such use-def coupling can occur when a method returns a value (in their example this value is a String representing the type of a book), which is given as parameter to another method (in their example a method which checks whether the type of a book is suitable to the person who wants to borrow it from the library). Even if the two methods are not directly coupled (there is no direct connection between them), if the values returned by the first method are changed, errors can be introduced into the second method. They propose an algorithm to detect for each variable the point where the variable

was initialized (to see the entities to which it is coupled) and implement this algorithm in an Eclipse plug-in, called ICD (Indirect Coupling Detector).

Yang and Tempero investigate in [33] the notion of *indirect dependence* and argue that it is an important criteria for evaluating modularity. The authors claim the importance of understanding *indirect coupling* (IC) due to its "hidden" nature. They highlight the importance of determining which forms of indirect coupling may be avoided, arguing that a system with high levels of avoidable indirect coupling is "unmodular" [33]. The same authors, Yang and Tempero extend in [35] their previous study and propose metrics which express the relationship between indirect coupling and maintainability. The proposed metrics are applied to existing Java applications.

While traditional coupling measures cannot be used for finding hidden dependencies, Poshyvanyk et al. [27] have presented how a conceptual coupling measure that considers identifier names, comments and other textual elements of code can be used for impact analysis and can find hidden dependencies as well. The study reports precision and recall around 20%.

Petrenko and Raylich [26] have introduced an interactive tool called JRipples which is useful for iterative impact analysis. The proposed tool does not discover HD, the software developer having the responsibility to correctly identify the hidden dependecies during impact analysis.

In case of large software systems, computing Abstract System Dependence Graphs can be expensive, so other approaches which are based on the order in which different methods are called (call trace) have been introduced: if a method is always called after another method, there might be a dependency (hidden or not) between the classes where these methods are, as presented in [31]. Vanciu and Rajlich [31] have proposed a dynamic technique for identifying hidden dependencies. It is based on computing "execute completely after" relations which are filtered based on pre- and postconditions that are generated dynamically. For evaluation, open source software systems like JUnit, Drawlets and Apache FtpServer are used. The authors show that hidden dependencies exist even in well-designed software, like the ones considered for evaluation. For the case studies used for evaluation, the technique proposed in [31] obtained a precision between 46% and 59% for discovering hidden dependencies.

Kirbas et al. [19] have investigated the influence of the evolutionary coupling on defect proneness. A positive correlation between evolutionary coupling and defect measures, such as number of defects and defect density, have been confirmed by numerical experiments performed for a large financial legacy software system. Two evolutionary coupling measures derived from modification requests (MR) have been used in this study.

He et al. [15] have proposed Coverage and Program Structure Slicing (CPSS) as a novel solution to fault localization. CPSS is based on Reverse Data Dependence Analysis Model and integrates Coverage Based Fault Localization (CBFL) and Program Slicing by analyzing the program structure. The proposed method has been experimentally proven to be more effective than existing related methods.

Kouroshfar et al. [20] have studied the effects of architecturally dispersed co-changes on software quality. It has been experimentally shown that the changes involving multiple architectural modules are more correlated with defects than the intra-module co-changes. The study corroborates the relevance of considering architecture in predicting software defects.

Akbarinasaji et al. [3] have proposed a suite of six metrics of logical dependency among source files in a software system. The impact of these metrics on defect prediction performance has been evaluated by applying two learning models, the Logistic Regression and the Naive Bayes, on three different software projects. The metrics have been used as features of the training data, their values being derived from the timestamp information in the change history of files. The experimental results have confirmed that, if the values of logical dependency are high, they significantly improve the performance of the defect prediction models.

Bell [6] has studied the influence of hidden dependencies identification on software testing. The author has shown that increasing the efficiency and the effectiveness of testing through a good knowledge of the hidden dependencies between tests improves the software reliability. In real software systems, there are hidden dependencies between tests, which makes the testing process harder. In such situations, the tests cannot be run in parallel, since they are not independent (i.e. a test outcome is influenced by the execution of other test). It has been shown in the software engineering literature [6] that these dependencies are often difficult and hidden from the software developers. Bell has developed a software system called VMVM for detecting different types of dependencies between tests and has used detected information to significantly reduce the testing time (with around 60% in average). VMVM is a Java implementation of a technique called Unit Test Virtualization, a technique which isolates the side-effects of each unit test from other tests. It is based on a hybrid static-dynamic analysis and automatically identifies the code segments that may create side-effects. These segments are isolated in a container similar to a virtual machine.

Due to the complexity of the HDI problem, there is a continuous interest in the software engineering literature to develop more performant detectors.

4. Discussion

The evaluation measure which is usually used for estimating the performance of a process that detects hidden dependencies is the *precision* of the detection [31]. The *precision* of a HDI process is computed as the percentage of dependencies that were correctly reported as *hidden*. Since the entire set of HDs is unknown, the *recall* measure is impractical in this context.

After the in-depth analysis of the related work we presented in Section 3, we can conclude that there are a number of limitations of the approaches existing in the literature for hidden dependencies identification.

Regarding the performance of the identification process, the existing approaches have moderate *precision* values: in [26] the precision ranges from 6% to 18%, [27] reports precision around 20%, while in [4], it ranges from 30% to 40%. An improvement of the performance of HDI is achieved in [31] which reports precision between 46% and 60%.

Besides, some existing approaches rely on historical data, which is not always available (and knowledge extracted from it cannot be used for other projects), or on the creation of different graphs which can be expensive for large systems.

Even if there are a lot of approaches existing in the literature in the direction of *impact analysis* and *hidden dependencies identification*, to the best of our knowledge, the applicability of *machine learning* methods has not been investigated yet. Due to their ability to unconver hidden patterns in data, we consider that *machine learning* models would be appropriate for detecting hidden dependencies in software projects and that this direction may provide valuable results in the field.

4.1. **Our approach.**

Our first objective to achieve the long-term goal of this research is to investigate how to improve *impact analysis* approaches. We are planning to reach this objective by developing new coupling measurements to improve the performance of estimating the impacts of future changes in software systems. We aim to capture in the coupling measures both the *structural* and *conceptual* aspects of coupling.

Our second research direction will be to propose *machine learning* methods for detecting *hidden dependencies* in software systems. As we deduced from reviewing the problem of *hidden dependencies identification*, none of the approaches from the literature use machine learning algorithms. Out of the existing approaches, using call trace information seems promising. We believe that *relational association rules* (RARs) [29] can be used to mine relevant patterns in the call traces. Based on our previous experience with *relational association rule mining*, we consider that RARs have the potential to improve the precision and recall values, since low values make the existing approaches

impractical to be used for real systems. Besides relational association rules, we will also investigate the applicability of unsupervised learning techniques, such as *clustering* or *self-organizing maps* [18].

In our view, the problem of HDI can be formalized as a *clustering* problem. *Clustering* [13] (also known as unsupervised classification) is able to differentiate groups of similar objects inside a given data set through detecting *hidden* patterns in data. Thus, we consider that a *clustering* approach may be useful in detecting *hidden dependencies*.

Let us consider that a software system S is represented as a set of *software entities*, $\mathcal{S} = \{e_1, e_2, \dots, e_n\}$. Depending on the granularity of the approach, a software entity $e_i \in \mathcal{S}$ can be a software component, an application class, a method or an attribute from a class, etc. The clustering approach we propose for HDI consists of three main steps and is depicted in Figure 5:

- **Data representation**. The *software entities* and the existing relationships between them (inheritance, dependency, aggregation, etc.) are extracted from the analyzed software system. Each software entity will be represented by a high-dimensional vector. The challenge will be to determine a set of *software metrics* relevant for deciding if a *hidden dependency* exists between two entities.
- **Grouping**. The set of entities extracted at the previous step are grouped in clusters using an *unsupervised learning* method (e.g. *clustering* [13] or *self-organizing map* [30]). The goal of this step is to obtain groups (clusters) which will contain software entities which depend on each other (considering both *direct* and *hidden* dependencies).
- **HD extraction**. The clusters obtained after the *Grouping* step will be filtered in order to remove the *direct* dependencies. The remaining entities from each cluster will provide a list of HDs.

Figure 5 contains a graphical representation of the solution we propose for *hidden dependencies identification*.

4.2. **Preliminary experimental results.** In this section we give some incipient experimental results which underline the effectiveness of using *unsupervised learning* for detecting software *dependencies*. We consider an experiment on an open source software framework, *Commons DbUtils* (version 1.3), a library consisting of a small set of classes which are designed to make working with JDBC easier [1]. It consists of 22 classes, placed in three packages:

- *default package* - contains the core classes and interfaces of the system.
- *handlers* - contains implementations for the *ResultSetHandler* interface from the default package.

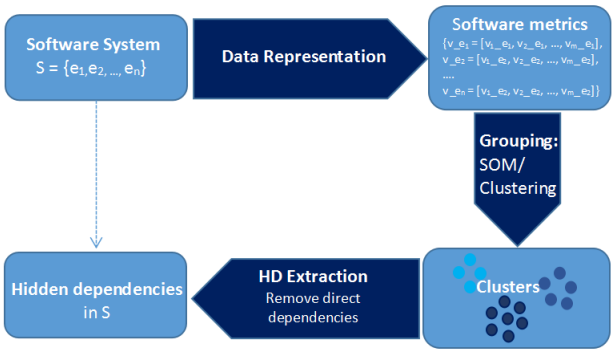

FIGURE 5. The proposed solution.

Package	Class Name
default	BasicRowProcessor(BRP), BeanProcessor (BP), DbUtils, ProxyFactory (PF), QueryLoader (QL), QueryRunner (QR), ResultSetHandler (RSH), ResultSetIterator (RSI), RowProcessor (RP)
handlers	AbstractKeyedHandler (AKH), AbstractListHandler (ALH) ArrayHandler (AH), ArrayListHandler (ALH), BeanHandler (BH), BeanListHandler (BLH), KeyedHandler (KH), ColumnListHandler (CLH), MapListHandler (MPH), MapHandler (MH), ScalarHandler (SH)
wrappers	SqlNullCheckedResultSet (SNCRS), StringTrimmedResultSet (STRS)

TABLE 1. Packages and classes in the DbUtils 1.3 system.

- *wrappers* - contains two wrappers for the *ResultSet* class from the *java.sql* package.

The exact classes from each package are presented on Table 1.

The application classes from DbUtils 1.3 are converted into a text corpus containing the elements of the implementation code (including comments, identifiers, etc.). Then, the corpus associated to the class is represented as a fixed-length feature vector of numerical values. These feature vectors are unsupervisedly learned using the implementation of *Paragraph Vector* (or `Doc2Vec`) offered by Gensim [2]. `Doc2Vec`, a model proposed by Le and Mikolov [21], is useful for expressing variable-length textual information as a fixed-length dense numeric vector (*paragraph vector*), being an alternative to common models such as bag-of-words and bag-of-n-grams. A first advantage of `Doc2Vec`

	AKH	ALH	AH	ALH	BRP	BH	BLH	BP	CLH	DU	KH	MH	MLH	PF	QL	QR	RSH	RSI	RP	SH	SNCRS	STR
AKH	1.000	0.654	0.711	0.714	0.463	0.758	0.769	0.416	0.632	0.533	0.728	0.785	0.769	0.475	0.427	0.115	0.668	0.572	0.834	0.597	0.091	0.439
ALH	0.654	1.000	0.668	0.736	0.360	0.765	0.779	0.323	0.503	0.287	0.301	0.731	0.726	0.425	0.264	0.127	0.853	0.521	0.669	0.477	0.368	0.606
AH	0.711	0.668	1.000	0.950	0.511	0.918	0.900	0.269	0.814	0.473	0.644	0.956	0.941	0.616	0.485	0.253	0.769	0.727	0.704	0.812	0.382	0.585
ALH	0.714	0.736	0.950	1.000	0.481	0.895	0.899	0.328	0.792	0.556	0.561	0.952	0.978	0.695	0.558	0.341	0.840	0.682	0.698	0.760	0.455	0.614
BRP	0.463	0.360	0.511	0.481	1.000	0.547	0.557	0.373	0.418	0.147	0.547	0.501	0.447	0.160	0.473	0.286	0.413	0.322	0.489	0.395	0.254	0.523
BH	0.758	0.765	0.918	0.895	0.547	1.000	0.979	0.384	0.825	0.360	0.628	0.915	0.897	0.589	0.414	0.116	0.771	0.547	0.760	0.822	0.342	0.592
BLH	0.769	0.779	0.900	0.899	0.557	0.979	1.000	0.380	0.787	0.408	0.579	0.899	0.902	0.610	0.398	0.042	0.774	0.548	0.769	0.781	0.330	0.582
BP	0.416	0.323	0.269	0.328	0.373	0.384	0.380	1.000	0.271	0.093	0.334	0.286	0.293	0.339	0.321	0.111	0.410	0.061	0.326	0.322	0.125	0.258
CLH	0.632	0.503	0.814	0.792	0.418	0.825	0.787	0.271	1.000	0.248	0.747	0.824	0.805	0.514	0.460	0.314	0.614	0.468	0.602	0.973	0.365	0.453
DU	0.533	0.287	0.473	0.556	0.147	0.360	0.408	0.093	0.248	1.000	0.236	0.533	0.590	0.563	0.293	0.402	0.430	0.434	0.517	0.211	0.175	0.347
KH	0.728	0.301	0.644	0.561	0.547	0.628	0.579	0.334	0.747	0.236	1.000	0.673	0.617	0.186	0.479	0.153	0.440	0.489	0.602	0.715	0.057	0.195
MH	0.785	0.731	0.956	0.952	0.501	0.915	0.899	0.286	0.824	0.533	0.673	1.000	0.969	0.592	0.488	0.299	0.808	0.718	0.734	0.802	0.366	0.580
MLH	0.769	0.726	0.941	0.978	0.447	0.897	0.902	0.293	0.805	0.590	0.617	0.969	1.000	0.682	0.516	0.337	0.804	0.658	0.713	0.767	0.372	0.553
PF	0.475	0.425	0.616	0.695	0.160	0.589	0.610	0.339	0.514	0.563	0.186	0.592	0.682	1.000	0.278	0.349	0.594	0.194	0.530	0.538	0.475	0.569
QL	0.427	0.264	0.485	0.558	0.473	0.414	0.398	0.321	0.460	0.293	0.479	0.488	0.516	0.278	1.000	0.303	0.375	0.310	0.290	0.381	0.221	0.230
QR	0.115	0.127	0.253	0.341	0.286	0.116	0.042	0.111	0.314	0.402	0.153	0.299	0.337	0.349	0.303	1.000	0.330	0.266	0.138	0.292	0.221	0.119
RSH	0.668	0.853	0.769	0.840	0.413	0.771	0.774	0.410	0.614	0.430	0.440	0.808	0.804	0.594	0.375	0.330	1.000	0.618	0.768	0.629	0.466	0.662
RSI	0.572	0.521	0.727	0.682	0.322	0.547	0.548	0.061	0.468	0.434	0.489	0.718	0.658	0.194	0.310	0.266	0.618	1.000	0.503	0.468	0.199	0.418
RP	0.834	0.669	0.704	0.698	0.489	0.760	0.769	0.326	0.602	0.517	0.602	0.734	0.713	0.530	0.290	0.138	0.768	0.503	1.000	0.592	0.145	0.483
SH	0.597	0.477	0.812	0.760	0.395	0.822	0.781	0.322	0.973	0.211	0.715	0.802	0.767	0.538	0.381	0.292	0.629	0.468	0.592	1.000	0.393	0.484
SNCRS	0.091	0.368	0.382	0.455	0.254	0.342	0.330	0.125	0.365	0.175	0.057	0.366	0.372	0.475	0.221	0.221	0.466	0.199	0.145	0.393	1.000	0.815
STRS	0.439	0.606	0.585	0.614	0.523	0.592	0.582	0.258	0.453	0.347	0.195	0.580	0.553	0.569	0.230	0.119	0.662	0.418	0.483	0.484	0.815	1.000

TABLE 2. The absolute values of cosine similarities between the classes from DBUtils 1.3.

over the traditional models is that it considers the semantics of the words or, more formally, the distance between the words [21]. Therefore, *private* will be closer to *protected* than to *boolean*. An additional advantage over bag-of-words is that it also takes into consideration the words order, at least in a small context.

In our experiment with DBUtils 1.3, we computed feature vectors consisting of 300 numerical features. We give in Table 2 the absolute values of the cosine similarities between all pairs of feature vectors.

Our focus is to test if an *unsupervised learning* model is able to capture the *coupling* relationship between the application classes thus avoiding to limit the definition of *coupling* to a predefined similarity function (like cosine similarity). A *self-organizing map* will be used in our experiment as an *unsupervised learning* model. SOMs [30] are a type of artificial neural network which are trained to provide a low-dimensional representation of the input space, called a *map* [10]. The main characteristic of a SOM is that it preserves the topological ordering of the input data, more exactly the input instances which are close to each other in the input space will also be close to each other on the output map.

The 22 application classes from DbUtils 1.3 are mapped on a SOM having a *torus* topology. For visualizing the SOM, the U-Matrix method [18] is used. Figure 6 illustrates the U-Matrix visualization of the SOM trained on the application classes from DbUtils 1.3. The darker regions on the U-Matrix represent data that are similar while the data falling in the lighter regions are dissimilar. Visualizing the U-Matrix for the resulting map, we observe three regions corresponding to the three packages presented in Table 1. The classes from the *default* package are displayed in *red*, those from the *handlers* package in *green*, while the third package *wrappers* is marked with *blue*.

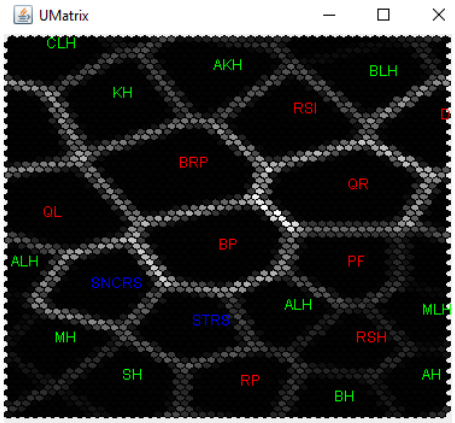

FIGURE 6. U-Matrix visualization.

Analyzing the U-matrix from Figure 6 we observe two application classes (*RowProcessor* and *ResultSetHandler*) which seem to be misplaced in the *handlers* package. But these two misplacements are explainable, considering the conceptual coupling measurement we used in our experiment. The *ResultSetHandler* class is conceptually coupled to the classes from the *handlers* package and this type of coupling is deduced from its source code. The *RowProcessor* class is close on the map to the *BeanHandler* class. Analyzing the source code of the *BeanHandler* class we found that it contains an attribute of type *RowProcessor*, which justifies their closeness on the map. Moreover, inspecting the source code of *RowProcessor* class, we observe that it operates with (Java) *beans* and this is expressed in its code (identifiers, comments, etc). Thus, the representation of the classes using `Doc2Vec` captured the conceptual relationship between the classes. We can conclude that the map depicted in Figure 6 empirically confirms our hypothesis that unsupervised *machine learning* models (the *self-organizing map*, in our case) are able to express *dependencies* (*conceptual*, in our case) between software entities. For capturing the direct coupling between software entities, we should consider not only the *conceptual* coupling, but also the *structural* one.

In our experiment we have focused only on *direct* dependencies, but we are confident that using an appropriate data representation (i.e. vectorial representation of the *application classes*), a SOM will be effective for depicting more complex dependencies (like *hidden dependencies*) from a software system. Further work will investigate different vectorial representations for the

software entities which are appropriate for capturing more complex software dependencies.

5. Conclusions and future work

This paper presented in detail the problem of identifying hidden dependencies in software systems, a problem of major importance during the maintenance and evolution of software systems. We discussed about evaluating the performance of the detection process and we identified the main limitations of the existing state-of-the-art approaches.

We proposed a new computational model based on clustering for the problem of *hidden dependencies identification*. Such a *machine learning* perspective has not been proposed in the literature so far. As further work we will investigate *software metrics* useful in the *Data representation* step from our approach, as well as different clustering algorithms useful in the *Grouping* step. Regarding the *impact analysis*, we target to develop coupling measurements which capture both the structural and the conceptual aspects of coupling.

Acknowledgments

This work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS−UEFISCDI, project number PN-II-RU-TE-2014-4-0082.

References

[1] Commons DbUtils. http://commons.apache.org/proper/commons-dbutils/index.html.
[2] RaRe TECHNOLOGIES. https://github.com/RaRe-Technologies/gensim.
[3] Shirin Akbarinasaji, Behjat Soltanifar, Bora Çağlayan, Ayse Basar Bener, Andriy Miranskyy, Asli Filiz, Bryan M. Kramer, and Ayse Tosun. A metric suite proposal for logical dependency. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*, WETSoM '16, pages 57–63, New York, NY, USA, 2016. ACM.
[4] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 432–441, New York, NY, USA, 2005. ACM.
[5] A. Beer and S. Mohacsi. Efficient test data generation for variables with complex dependencies. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 3–11, April 2008.
[6] Jonathan Bell. *Making Software More Reliable by Uncovering Hidden Dependencies*. PhD thesis, Graduate School of Art and Sciences, Columbia University, 2016.
[7] Lionel C. Briand, Juergen Wuest, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 475–482, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Daniel Conte de Leon and Jim Alves-Foss. Hidden implementation dependencies in high assurance and critical computing systems. *IEEE Trans. Softw. Eng.*, 32(10):790–811, October 2006.

[9] D. Conte de Leon and J. Alves-Foss. Hidden implementation dependencies in high assurance and critical computing systems. *IEEE Transactions on Software Engineering*, 32(10):790–811, Oct 2006.

[10] N. Elfelly, J.-Y. Dieulot, and P. Borne. A neural approach of multimodel representation of complex processes. *International Journal of Computers, Communications & Control*, III(2):149–160, 2008.

[11] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–198, Washington, DC, USA, 1998. IEEE Computer Society.

[13] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[14] Ahmed E. Hassan and Richard C. Holt. Predicting change propagation in software systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Hui He, Dongyan Zhang, Min Liu, Weizhe Zhang, and Dongmin Gao. A coverage and slicing dependencies analysis for seeking software security defects. *The Scientific World Journal*, 2014:1–10, 2014.

[16] Jakob Jenkov. Understanding Dependencies. Technical report, Tech and Media Labs, 2014.

[17] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, March 2007.

[18] S. Kaski and T. Kohonen. Exploratory data analysis by the self-organizing map: Structures of welfare and poverty in the world. In *Neural Networks in Financial Engineering. Proceedings of the Third International Conference on Neural Networks in the Capital Markets*, pages 498–507. World Scientific, 1996.

[19] Serkan Kirbas, Alper Sen, Bora Caglayan, Ayse Bener, and Rasim Mahmutogullari. The effect of evolutionary coupling on software defects: An industrial case study on a legacy system. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 6:1–6:7, New York, NY, USA, 2014. ACM.

[20] Ehsan Kouroshfar, Mehdi Mirakhorli, Hamid Bagheri, Lu Xiao, Sam Malek, and Yuanfang Cai. A study on the role of software architecture in the evolution and quality of software. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 246–257, Piscataway, NJ, USA, 2015. IEEE Press.

[21] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. *CoRR*, abs/1405.4053, 2014.

[22] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 41–50, New York, NY, USA, 2011. ACM.

[23] Thomas M. Mitchell. *Machine learning*. McGraw-Hill, Inc. New York, USA, 1997.
[24] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.
[25] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.
[26] Maksym Petrenko and Vclav Rajlich. Variable granularity for improving precision of impact analysis. In *ICPC*, pages 10–19. IEEE Computer Society, 2009.
[27] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Softw. Engg.*, 14(1):5–32, February 2009.
[28] Vaclav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 84–91, Washington, DC, USA, 1997. IEEE Computer Society.
[29] Gabriela Serban, Alina Câmpan, and Istvan Gergely Czibula. A programming interface for finding relational association rules. *International Journal of Computers, Communications & Control*, I(S.):439–444, June 2006.
[30] Panu Somervuo and Teuvo Kohonen. Self-organizing maps and learning vector quantization for feature sequences. *Neural Processing Letters*, 10:151–159, 1999.
[31] R. Vanciu and V. Rajlich. Hidden dependencies in software systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.
[32] Lee White, Khaled Jaber, Brian Robinson, and Václav Rajlich. Extended firewall for regression testing: An experience report. *J. Softw. Maint. Evol.*, 20(6):419–433, November 2008.
[33] H. Y. Yang and E. Tempero. Indirect coupling as a criteria for modularity. In *Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM '07. First International Workshop on*, pages 10–10, May 2007.
[34] Hong Yul Yang, E. Tempero, and R. Berrigan. Detecting indirect coupling. In *2005 Australian Software Engineering Conference*, pages 212–221, March 2005.
[35] Hong Yul Yang and Ewan Tempero. Measuring the strength of indirect coupling. In *Proceedings of the 2007 Australian Software Engineering Conference*, ASWEC '07, pages 319–328, Washington, DC, USA, 2007. IEEE Computer Society.
[36] Hong Yul Yang, Ewan Tempero, and Rebecca Berrigan. Detecting indirect coupling. In *Proceedings of the 2005 Australian Conference on Software Engineering*, ASWEC '05, pages 212–221, Washington, DC, USA, 2005. IEEE Computer Society.
[37] Zhifeng Yu and V. Rajlich. Hidden dependencies in program comprehension and change propagation. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 293–299, 2001.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

E-mail address: {istvanc,gabis,marianzsu}@cs.ubbcluj.ro,mdir1308@scs.ubbcluj.ro