Measuring the Structural Similarity between Source Code Entities

Ricardo Terra*, João Brunet†, Luis Miranda*, Marco Túlio Valente*,
Dalton Serey†, Douglas Castilho*, and Roberto Bigonha*

*Universidade Federal de Minas Gerais, Brazil
Email: {terra,luisfmiranda,mtov,douglas.castilho,bigonha}@dcc.ufmg.br

†Universidade Federal de Campina Grande, Brazil
Email: {jarthur,dalton}@dsc.ufcg.edu.br

***Abstract*—Similarity coefficients are widely used in software engineering for several purposes, such as identification of refactoring opportunities and system remodularizations. Although the literature provides several similarity coefficients that vary on the computing strategy, there is a tendency among researchers to make habitual use of certain coefficients that others in their field are using. Consequently, some approaches might be using an inadequate coefficient for their purpose. In this paper, we conduct a quantitative study that compares 18 coefficients to identify which one is the most appropriate in determining where a class should be located. Our evaluation contemplates 111 open source systems from Qualitas Corpus, which totalizes more than 70,000 classes. As a result, we observed that Jaccard—one of the most used coefficients in our area—has not presented the best results. While Jaccard correctly indicated the suitable module to 22% of the classes, other coefficients were able to indicate 60%.**

I. Introduction

Similarity coefficient measures the degree of correspondence between two entities according to an established criterion. This concept is widely used in software engineering area for several different purposes, such as identification of refactoring opportunities [1]–[3] and system remodularizations [4], [5]. For example, Fokaefs et al. employ a well-known similarity coefficient—named *Jaccard* [6]—to measure similarity between methods of a given class in order to recommend a *Extract Class* refactoring for those methods with low similarity. As another example, Simon et al. also employed *Jaccard* coefficient to propose a set of cohesion metrics that help developers to identify refactoring opportunities, such as *Move Method* and *Move Attribute* [3].

The literature is prolific and provides several similarity coefficients that vary on their computing strategy. Nevertheless, there are few works comparing similarity coefficients using structural dependencies as source of information [4]. This lack of knowledge may lead researchers to choose an inadequate coefficient to this purpose, once there is a tendency among researchers to make habitual use of certain coefficients that others in their field are using, even without sound scientific or empirical reasons [7].

In this paper, we conduct a quantitative study that compares 18 coefficients to identify which one is the most appropriate in determining where a class should be located. We computed the similarity between classes and packages of 111 open source systems from Qualitas Corpus [8]. From our results, we can point out three main findings:

1) Structural dependencies are indeed precise enough to determine where a class should be located. In our evaluation, we achieved an overall precision of 80% in indicating the correct package of a class up to rank 3.

2) Considering the dependency type or the multiplicity of dependencies does not improve the overall precision. Our results show that simply relying on the existence of dependencies between two entities—i.e., without considering the dependency type and multiplicity—achieves the best precision results.

3) *Jaccard*—one of the most used coefficients in our area— has not presented the best results. While *Jaccard* indicated the correct package to only 22% of the classes, other coefficients—such as *Relative Matching*, *Kulczynski*, and *Russell and Rao*—were able to indicate to slightly over 60%.

A usual problem that developers face during software refactoring or remodularization is to indicate the suitable package in which a particular class should be located. In this context, our findings might improve software engineering approaches that need to determine the suitable package of a class. As aforementioned, *Jaccard* is not the most precise coefficient in the context of measuring the similarity between classes and packages using structural dependencies as source of information.

The remainder of this paper is structured as follows. Section II provides a description of similarity coefficients. Section III describes strategies for extracting structural dependencies from a class. Section IV presents and discusses results on comparing our strategies and coefficients in 111 real-world systems. Finally, Section V presents related work and Section VI concludes the paper.

II. Similarity Coefficients

Table I shows 18 similarity coefficients that we have evaluated to determine the most appropriated one in the context of measuring similarity among classes [7], [9]. To calculate these coefficients, we assume that a given source code entity (method, class, or package) is represented by the dependencies it establishes with other types. Therefore, the measure of the structural similarity between two source code entities i and j (i.e., S_{ij}) considers the following variables:

- $\boldsymbol{a}$ = the number of dependencies on both entities,
- $\boldsymbol{b}$ = the number of dependencies on entity $\boldsymbol{i}$ only,
- $\boldsymbol{c}$ = the number of dependencies on entity $\boldsymbol{j}$ only, and
- $\boldsymbol{d}$ = the number of dependencies on neither of the entities.

For instance, *Jaccard*—one of the simplest and most used coefficient in our field—is defined by:

$$S_{ij} = \frac{a}{a+b+c} \quad (1)$$

Basically, *Jaccard* indicates maximum similarity when two entities have identical dependencies, i.e., when $b = c = 0$ and thus $S_{ij} = 1.0$. On the other hand, it indicates minimum similarity when there are no dependencies in common, i.e., when $a = 0$ and thus $S_{ij} = 0.0$.

```
class Bar extends X {
    A a;
    B b;

    exampleBar(D d){
        a.f();
        d.g();
    }
}
```

```
class Foo extends X {
    B b;
    G g;

    exampleFoo(E e){
        e.j();
        new A().f()
    }
}
```

Code 1. Hypothetical classes to explain the measurement of similarity

As an illustrative example, Code 1 presents two hypothetical classes. In order to measure the similarity between `Bar` and `Foo`, we first determine the value of the variables a, b, c, and d. In this example: $a = 3$ since both classes rely on `A`, `B`, and `X`; $b = 1$ since only `Bar` relies on `D`; $c = 2$ since only `Foo` relies on `E` and `G`; and $d = 3$ since none establishes dependencies with three other classes of the system (namely `C`, `F`, and `Y`). Next, we choose a similarity coefficient and solve the formula. For example, the similarity between `Bar` and `Foo` using *Jaccard* results in 0.5, whereas using *Phi* decreases to 0.35 or using *Kulczynski* increases to 0.675.

Each coefficient has a unique property that differs it from others. For example, while *Jaccard* does not consider what the both entities do not have in order to compute their similarity (variable d), *Simple matching* and 10 other coefficients contemplate it. The *Yule* and *Hamann* coefficients are mathematically related. Although both have the same variables in their numerators and denominators, *Hamann* relates the variable by addition whereas *Yule* relates them by multiplication.

As another example, *Sorenson*[1] gives twice the weight to what the entities have in common (variable a), while the *Rogers and Tanimoto* coefficient gives twice the weight to what each entity has independently (variables b and c). Except for the d variable in the denominator, *Russell and Rao* resembles *Jaccard*. On the other hand, the *Sokal and Sneath* coefficient, which is quite similar to *Simple matching*, reduces the importance of what each entity has independently (variables b and c) by half.

Kulczynski and *Sokal and Sneath 4* are based on conditional probability. *Kulczynski* assumes that a characteristic is present in one item, given that it is present in the other, whereas the *Sokal and Sneath 4* coefficient assumes that a characteristic in one item matches the value in the other. Finally, *Relative Matching* considers a set of similarity properties such as no mismatch, minimum match, no match, complete match, and maximum match.

[1] *Sorenson* is also referred on the literature as *Czekanowski* or *Dice*.

TABLE I
General Purpose Similarity Coefficients

Coefficient	Definition S_{ij}	Range
1. Jaccard	$a/(a+b+c)$	0–1*
2. Simple matching	$(a+d)/(a+b+c+d)$	0–1*
3. Yule	$(ad-bc)/(ad+bc)$	-1–1*
4. Hamann	$[(a+d)-(b+c)]/[(a+d)+(b+c)]$	-1–1*
5. Sorenson	$2a/(2a+b+c)$	0–1*
6. Rogers and Tanimoto	$(a+d)/[a+2(b+c)+d]$	0–1*
7. Sokal and Sneath	$2(a+d)/[2(a+d)+b+c]$	0–1*
8. Russell and Rao	$a/(a+b+c+d)$	0–1*
9. Baroni-Urbani and Buser	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+(ad)^{\frac{1}{2}}]$	0–1*
10. Sokal binary distance	$[(b+c)/(a+b+c+d)]^{\frac{1}{2}}$	0*–1
11. Ochiai	$a/[(a+b)(a+c)]^{\frac{1}{2}}$	0–1*
12. Phi	$(ad-bc)/[(a+b)(a+c)(b+d)(c+d)]^{\frac{1}{2}}$	-1–1*
13. PSC	$a^2/[(b+a)(c+a)]$	0–1*
14. Dot-product	$a/(b+c+2a)$	0–1*
15. Kulczynski	$\frac{1}{2}[a/(a+b)+a/(a+c)]$	0–1*
16. Sokal and Sneath 2	$a/[a+2(b+c)]$	0–1*
17. Sokal and Sneath 4	$\frac{1}{4}[a/(a+b)+a/(a+c)+d/(b+d)+d/(c+d)]$	0–1*
18. Relative Matching	$[a+(ad)^{\frac{1}{2}}]/[a+b+c+d+(ad)^{\frac{1}{2}}]$	0–1*

The symbol "$*$" denotes the maximum similarity.

III. Strategies

In order to measure the similarity between two source code entities, we assume that a given source code entity (a class or a package, in this paper) is represented by the *structural dependencies* it establishes with other types. We also distinguish the type of the dependency, i.e., whether a given dependency was established by accessing methods and fields (`access`), declaring variables (`declare`), creating objects (`create`), extending classes (`extend`), implementing interfaces (`implement`), throwing exceptions (`throw`), or using annotations (`useannotation`). Structural dependencies are extracted from the source code using a function named $Deps(E, S)$. Basically, this function returns `E`'s dependencies according to a strategy `S`. A strategy is a pair $[\texttt{C}, \texttt{D}]$ that defines the *collection*[2] and the *data information* to be employed in the extraction. The collection `C` can assume one of the following values:

[2] We use the generic term "*collection*" when we do not need to be specific about the kind of structure (set or multiset) under consideration.

1) *set*: a collection that contains no duplicated elements. In other words, if a class establishes more than one dependency to `java.sql.Statement`, it considers only one.

2) *multiset*: a generalization of the notion of set in which elements are allowed to appear more than once. For instance, if a class establishes three dependencies to `java.sql.Statement`, we actually consider all of them.

The data information `D` can assume one of the following values:

1) *target type* (*tt*): in this case the extraction function returns a collection of target types that the entity establishes dependencies with. Thus, an element is a single $[\texttt{T}]$, denoting the existence of at least one dependency between the entity under analysis and `T`.

2) *target and dependency type* (*dtt*): in this case the extraction function returns a collection whose elements are pairs $[\texttt{dt}, \texttt{T}]$, denoting the existence of a dependency of type `dt` between the class under analysis and `T`.

```
public class Bar {
    public void foo (Date d){
        if (d == null){
            d = new Date();
        } else {
            new Date()
        }
    }
}
```

Code 2. An example class that explains our strategies

As an illustrative example, consider the class presented in Code 2. The collection returned by function $Deps(\texttt{Bar}, \texttt{S})$ differs according to the strategy `S` employed. More specifically, the following calls (and respective results) are possible:

$$Deps(\texttt{Bar}, [\texttt{set}, \texttt{tt}]) = \{\texttt{Date}\}$$
$$Deps(\texttt{Bar}, [\texttt{set}, \texttt{dtt}]) = \{[\texttt{declare}, \texttt{Date}], [\texttt{create}, \texttt{Date}]\}$$
$$Deps(\texttt{Bar}, [\texttt{mset}, \texttt{tt}]) = \{\texttt{Date}, \texttt{Date}, \texttt{Date}\}$$
$$Deps(\texttt{Bar}, [\texttt{mset}, \texttt{dtt}]) = \{[\texttt{declare}, \texttt{Date}], [\texttt{create}, \texttt{Date}], [\texttt{create}, \texttt{Date}]\}$$

Strategies that rely on *target and dependency type* may be particularly important when the classes of the system rely on types differently according to their location. For example, a factory method that creates a `DTO` (Data Transfer Object) and a logic presentation method that handles a `DTO` may not be similar. As another example, a class that implements `java.io.Serializable` and a method that declares `java.io.Serializable` may also not be similar. Although this strategy clearly performs better in particular cases, our evaluation is concerned with the overall precision.

Last but not least, the set of dependencies of a package `Pkg` is calculated by the union of the set of the dependencies of its classes as follows:

$$Deps(\texttt{Pkg}, \texttt{S}) = \bigcup_{C_i \in Pkg} Deps(\texttt{C}_\texttt{i}, \texttt{S})$$

IV. Evaluation

A. Research Questions

We designed a study to address the following overarching research questions:

RQ #1 – Are structural dependencies precise enough to indicate whether a class is located in the correct package?

RQ #2 – Considering the multiplicity of dependencies—i.e., a multiset rather than a set—improves the overall precision?

RQ #3 – Considering the dependency type—i.e., representing a target dependency as a pair $[\texttt{dt}, \texttt{T}]$ rather than only a single type $[\texttt{T}]$—improves the overall precision?

RQ #4 – Which coefficient is the most suitable to measure the similarity among classes of object oriented systems?

B. Target Systems

Our evaluation relies on the Qualitas Corpus[3], which is a collection of software systems intended to be used for empirical studies of code artifacts [8]. In its current version, the corpus includes the source code of many popular systems, such as JRE, Eclipse, NetBeans, and Apache Tomcat. Table II summarizes information about our data set.

TABLE II
Qualitas Corpus

# systems	111
# total of packages	6,841
# total of analyzed classes	71,823

It is worth noting that our data set is large and heterogeneous, ranging from text processors and small frameworks to complete IDEs and virtual machines.

C. Major Assumption

We made the following assumption due to the infeasibility in obtaining a 100% accurate oracle for thousands of classes.

"In order to conduct our experiment, we assume that every class under analysis is in its right package."

Therefore, similarity coefficients should indicate the *current* package of the class as its *most suitable one*.

D. Methodology

To provide answers to our research questions, we performed the following tasks:

1) *Setup*: First, we have set all system up, i.e., we imported and compiled the 111 projects from Qualitas Corpus.

2) *Data Extraction*: Second, we extracted the structural dependencies of classes using the four possible strategies described in Section III, i.e., $[\texttt{set}, \texttt{tt}]$, $[\texttt{set}, \texttt{dtt}]$, $[\texttt{mset}, \texttt{tt}]$, and $[\texttt{mset}, \texttt{dtt}]$.

[3]Qualitas Corpus v20120401. Available at: http://qualitascorpus.com.

3) *Comparative Analysis*: Third, we have measured the similarity using all coefficients described in Section II. The coefficients were applied to measure the similarity between pairs $[\texttt{class}, \texttt{package}]$ from our corpus.

4) *Qualitative Analysis*: Last, we have conducted a qualitative analysis in order to answer our research questions.

E. Experimental Setup

In order to conduct this experiment, the following policies have been proposed:

1) We have disregarded the class under analysis while searching for its right location. For example, when measuring the similarity between a class `A` and its package `Pkg`, we actually consider its own package `Pkg` as being $\texttt{Pkg} - \{\texttt{A}\}$.

   Thereupon, we have not sought the suitable location of classes whose package contains only such class. For example, assume that package `Pkg` contains only class `A`. The measure of similarity will be unfair because $Deps(\texttt{Pkg} - \{\texttt{A}\}, \texttt{S}) = \phi$.

2) We have disregarded a particular class $\texttt{C}_\texttt{i}$ when $|Deps(\texttt{C}_\texttt{i}, \texttt{S})| < 5$, i.e., we have not evaluated classes that establish less than five dependencies. These classes contain too little information to make any inference based on their structural dependencies.

3) We have not evaluated test classes, since most of the systems organize their test classes on a single package. Consequently, the test package contains classes related to different parts of the system—i.e., they are not structurally related—which certainly reduces the precision of any approach based on structural dependencies.

4) We have filtered trivial dependencies, such as those established with primitive and wrappers types (e.g., `int` and `java.lang.Integer`), `java.lang.String`, and `java.lang.Object`. Since virtually all classes establish dependencies with these types, they do not actually contribute for the measure of similarity. This decision is quite similar to text retrieval systems that exclude stop words because they are rarely helpful in describing the content of a document.

F. Results

Figure 1 illustrates the overall precision for each coefficient regarding the four analyzed strategies. The overall precision is defined by the ratio between the number of classes that have their location (package) correctly predicted by the similarity coefficient and the total number of analyzed classes. We have also provided the Top 1, 2, and 3 ranking, which stands for the position of the correct package of a class. As an example, considering strategy $[\texttt{set}, \texttt{tt}]$, the *Relative Matching* precision has reached 60% on Top 1, 72% on Top 2, and 78% on Top 3. In other words, it means that *Relative Matching* located the correct package of a class 60% on the first position of its ranking, 12% on the second position, and 6% on the third position.

Before we provide answers to our research questions, it is worth noting that many similarity coefficients presented very similar (mostly identical) results. In fact, the Spearman correlation among these coefficients was very close to 1, which allowed us to group them. The multiple correlation among *Simple Matching*, *Hamann*, *Rogers and Tanimoto*, *Sokal and Sneath*, and *Sokal Binary Distance* presented lowest correlation value of 0.999994. Similarly, *Jaccard*, *Sorenson*, *Dot-product*, and *Sokal and Sneath 2* presented lowest correlation value of 0.999999. Finally, *Ochiai* and *PSC* presented correlation equal to 0.998251. These results explain why there is no variance in the ranks within the same group.

Next, we answer our research questions based mainly on Figure 1. In all answers, our data interpretation always considers the Top 3 ranking—when not stated differently.

RQ #1: *Are structural dependencies precise enough to indicate whether a class is located in the correct package?*

Yes. As can be observed in Figure 1, there are coefficients that achieved a high precision to determine the package where a class should be located. In particular, *Relative Matching*, *Kulczynski*, *Russell and Rao*, and *Sokal and Sneath 4* indicate, in the worst scenario, over than 70% of precision.

RQ #2: *Considering the multiplicity of dependencies—i.e., a multiset rather than a set—improves the overall precision?*

No. Figure 1 shows that strategies that use the traditional set ($[\texttt{set}, \texttt{tt}]$ and $[\texttt{set}, \texttt{dtt}]$) perform better than an equivalent multiset-based strategy for all coefficients. The only exception is the *Russell And Rao* coefficient, which presented results slightly better for the $[\texttt{mset}, \texttt{tt}]$ strategy. More important, if we consider only Top 1, a traditional set-based strategy performs better than multiset-based one for all coefficients.

RQ #3 *– Considering the dependency type—i.e., representing a target dependency as a pair* $[\texttt{dt}, \texttt{T}]$ *rather than only a single type* $[\texttt{T}]$*—improves the overall precision?*

No. On one hand, Figure 1 shows that multiset-based data (i.e., $[\texttt{mset}, \texttt{tt}]$ and $[\texttt{mset}, \texttt{dtt}]$) presents very similar results for all coefficients. It is expected since the extracted collection is very similar. For instance, assume a collection `A` extracted using strategy $[\texttt{mset}, \texttt{dtt}]$ and a collection `B` extracted using strategy $[\texttt{mset}, \texttt{tt}]$. If $\texttt{A(i)} = [\texttt{access}, \texttt{Foo}]$ for an index `i`, then $\texttt{B(i)} = [\texttt{Foo}]$.

On the other hand, analyzing set data, we can observe that $[\texttt{set}, \texttt{tt}]$ provides better results for all coefficients, except for *Russell and Rao* and *Sokal and Sneath 4* that presented results slightly better using the dependency type ($[\texttt{set}, \texttt{dtt}]$).

From now on, our discussion only considers strategy $[\texttt{set}, \texttt{tt}]$, since we have demonstrated that the use of multiset

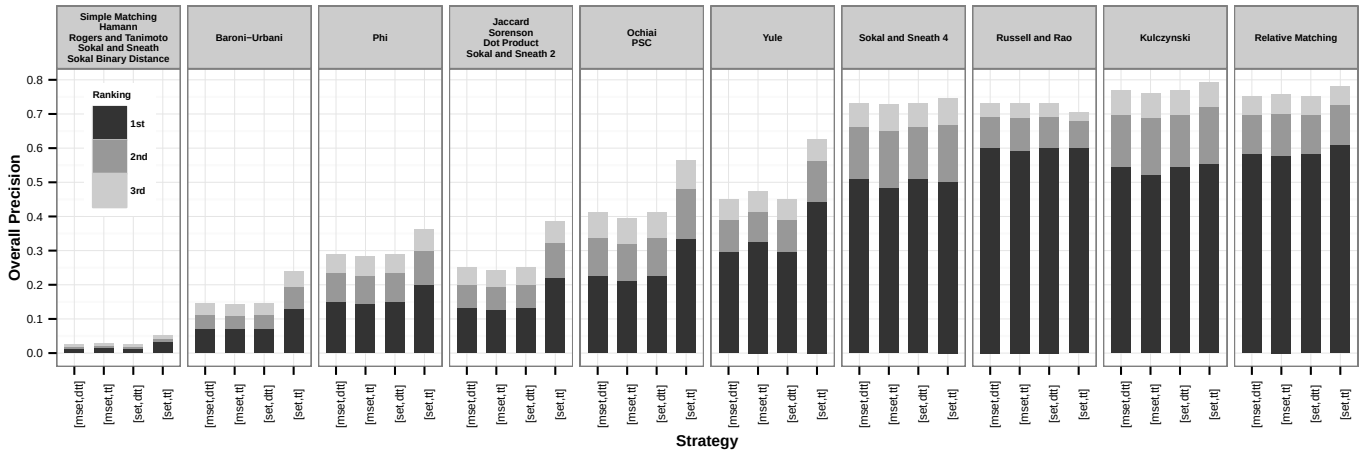

Fig. 1. Top 3 ranking of similarity coefficients using all strategies

(`mset`) and dependency type (`dtt`) does not actually improve the overall precision.

RQ #4: *Which coefficient is the most suitable to measure the similarity among classes of object oriented systems?*

Relative Matching*, *Kulczynski*, and *Russell and Rao. These coefficients have reached the highest precision values in our study. As can be observed in Figure 1, *Relative Matching* (60.83%) and *Russell and Rao* (60.27%) achieved the highest similarity values of the Top 1, and *Relative Matching* (72.78% and 78.18%) and *Kulczynski* (72.15% and 79.23%) of the Top 2 and 3.

As the central finding of our study, these three coefficients significantly outperform *Jaccard*—one of the most used similarity coefficients. While *Jaccard* indicated the correct package to 22% (Top 1) and 39% (Top 3) of the classes, *Relative Matching*, *Kulczynski* and *Russell and Rao* were able to indicate the correct package to 60% (Top 1) and 79% (Top 3).

To better explain this behavior, we anecdotally analyzed some systems to understand the influence of the variables a, b, c, and d (see Section II) on their precision. We performed this analysis by plotting each variable against the ranking.[4] Our major finding regards to the fact that large packages negatively influence *Jaccard* and other coefficients that presented very low precision (e.g., *Simple Matching*). Usually, large packages imply a large difference between c and d, which negatively impacts the precision of certain coefficients like *Jaccard*. On the other hand, by their nature, this scenario does not influence *Relative Matching*, *Kulczynski*, and *Russell and Rao*. It explains why these coefficients have presented the best results on both small and large packages.

G. Supplementary Results

Figure 2 illustrates the Top N ranking of every coefficient using strategy $[\texttt{set}, \texttt{tt}]$. In contrast to Figure 1 that displays only the Top 3, it displays the full distribution of the ranks

[4]Due to space constraints, we have not graphically presented this analysis.

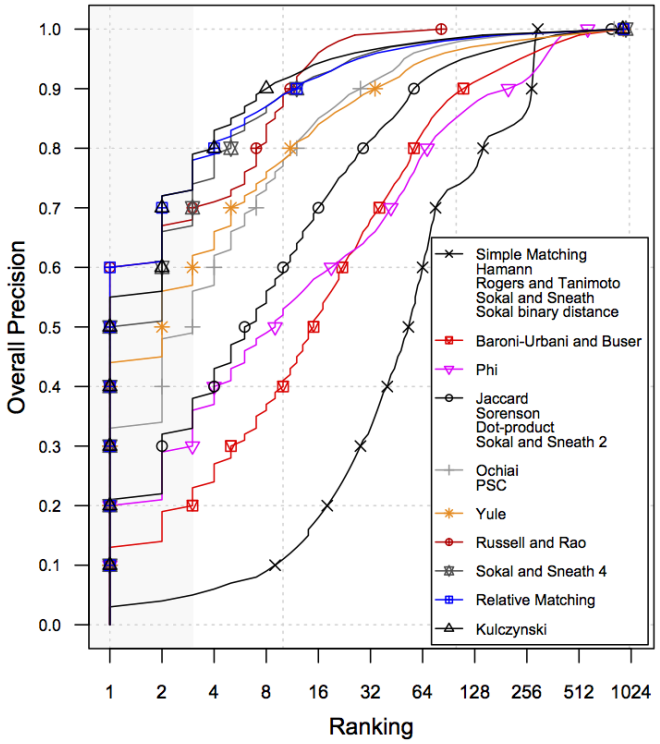

Fig. 2. General ranking using strategy $[\texttt{set}, \texttt{tt}]$

until full coverage (i.e., precision of 1.0). As a second relevant finding from our study, Figure 2 demonstrates that there is no coefficient that drastically improves its precision right after the top 3 ranking (e.g., Top 4 or Top 5). This behavior reinforces our decision in using Top 3. As can also be observed in Figure 2, *Russell and Rao* achieved precision of 1.0 in the rank 79. It means that the suitable package of a class was detected, in the worst case, on its 79th position. This result is quite relevant, since the other coefficients only achieved precision of 1.0 from the rank 347.

Since our analysis so far has considered the overall precision, we also analyzed the results of each coefficient per

system. Figure 3 summarizes the number of systems in which a particular coefficient has presented the best result (i.e., better identified the correct package of a class). For instance, *Relative matching* has better determined correct modules to *Eclipse* classes, whereas the *Russell and Rao* coefficient has behaved better to *ArgoUML* classes. Furthermore, we have not graphically presented some coefficients (e.g., *Simple Matching* and *Jaccard*) because they have not presented the best result for any system.

As can be observed in Figure 3, *Relative Matching*, *Kulczynski*, and *Russell and Rao* have presented the best results for most systems, which reinforces our claim that these coefficients are the most suitable ones to measure the similarity among classes in object-oriented systems.

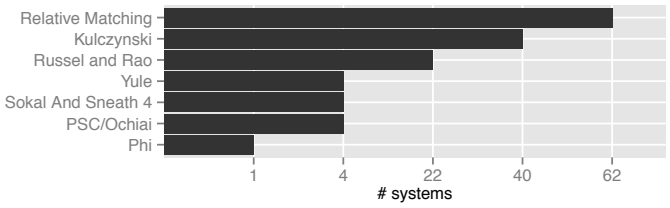

Fig. 3. # systems in which a particular coefficient presented the best result

H. Threats to Validity

We must state at least one major threat to the conclusion validity of the reported evaluation. Our experiment assumes that every class is in the right location. Although there might be misplaced classes, we rely on a stable and trustworthy collection of systems.

V. Related Work

There are few research works that compares similarity coefficients using structural dependencies as source of information [4]. Despite this lack of knowledge, similarity coefficients have been widely used for several different purposes [1]–[3], [5]. For example, the *Jaccard* distance between a method and a class is employed to support the automated identification of Feature Envy bad smells [1], [2]. Similarly, Simon et al. employ *Jaccard* distance to analyze similarity between classes and to identify refactoring opportunities [3]. They have proposed a cohesion metric based on *Jaccard* distance in order to suggest refactorings that improve the measurements of the metric. Our results suggest that the precision of the aforementioned approaches may be improved by using other coefficients that outperform *Jaccard* (e.g., *Relative Matching* and *Kulczynski*).

Fokaefs et al. employed *Jaccard* coefficient to develop a clustering method to suggest *Extract Class* refactorings for those entities with low level of similarity [5]. However, it is not clear in their paper how the authors handle structural dependencies to measure similarity among classes.

VI. Conclusion

First, we take the position that the choice of a similarity coefficient should not continue to be made without well-founded reasons. To address this shortcoming, we conducted a quantitative study that compares 18 coefficients to identify which one is the most appropriate in determining where a class should be located. As the major result, we observed that *Jaccard*—one of the most used coefficients in our area—has not presented the best results. While *Jaccard* indicated the correct package to only 22% of the classes, other coefficients—such as *Relative Matching*, *Kulczynski*, and *Russell and Rao*—were able to indicate to slightly over 60%.

Next, we have observed that the simplest strategy to extract structural dependencies from a class—set with only types ($[\texttt{set}, \texttt{tt}]$)—is indeed the best one. Stated differently, considering multisets of dependencies (`mset`) or considering also the dependency type (`dtt`) does not improve the overall precision.

Plans for future work include: (i) a sensitivity analysis of the factors `a`, `b`, `c`, and `d` in the ranking to statistically explain the behavior of each coefficient; (ii) an investigation of the impact on the results when measuring similarity of each pair $[\texttt{class}, \texttt{class}]$ and hence the similarity between a class `C` and a package `Pkg` will be calculated considering the average of the resulting similarity between `C` and `Pkg`'s classes; (iii) the extension of our comparative study to determine the most suitable class for a method; and (iv) the development of a tool that points out misplaced methods or classes.

Furthermore, we also have plans to use the main findings of the present study in the implementation of `ArchFix` [10], the recommendation system we are currently proposing to help developers to reverse software architecture erosion.

Acknowledgments: Our research has been supported by CAPES, FAPEMIG, and CNPq.

References

[1] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 99, pp. 347–367, 2009.

[2] ——, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[3] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *5th European Conference on Software Maintenance and Reengineering (CSMR)*, 2001, pp. 30–38.

[4] R. Naseem, O. Maqbool, and S. Muhammad, "Improved similarity measures for software clustering," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 45–54.

[5] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1037–1039.

[6] I. H. Moghadam and M. Ó. Cinnéide, "Automated refactoring using design differencing," in *15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 43–52.

[7] H. C. Romesburg, *Cluster Analysis for Researchers*. Lifetime Learning Publications, 1981.

[8] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas Corpus: A curated collection of Java code for empirical studies," in *17th Asia Pacific Software Engineering Conference (APSEC)*, 2010, pp. 336–345.

[9] B. S. Everitt, S. Landau, M. Leese, and D. Stahl, *Cluster Analysis*, 5th ed. Wiley, 2011.

[10] R. Terra, M. T. Valente, K. Czarnecki, and R. Bigonha, "Recommending refactorings to reverse software architecture erosion," in *16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track*, 2012, pp. 335–340.