University of Sindh Journal of Information and Communication Technology (USJICT)
Volume 3, Issue 1, January 2019
ISSN-E: 2523-1235, ISSN-P: 2521-5582
Website: http://sujo2.usindh.edu.pk/index.php/USJICT/

Measuring The Distance Between High-Level Models In A Reengineering Process.

Ignacio Cassol[1,a], Ignacio Berdiñas[1,b], Gabriela Arévalo[2,c]

[1]Facultad de Ingeniería, Universidad Austral, LIDTUA (CIC), Argentina
[2]Departamento de Ciencia y Tecnología, Universidad Nacional de Quilmes, Argentina
[a]icassol@austral.edu.ar, [b]ignacio.berdinas@ing.austral.edu.ar, [c]garevalo@unq.edu.ar

Abstract: When refactoring high-level models, measuring the differences between the original and the refactored model helps the designers know how the original model was modified and if the transformation added more complexity or/and improved the model. In our previous work, we developed the M2K methodology that parses legacy C code, maps it in a high-level model to represent the domain concepts and proposes a refactored model to improve the mapped design. Based on both models, we propose a distance to indicate, from the domain viewpoint, if the original identified concept keeps the same structure or, conversely, if the refactorings modify the concepts represented in the original model. Our approach is based on models generated through the M2K methodology and does not take into account syntactical variations between models. To show the applicability and the validation of our approach, firstly we show how we apply it on a trivial case study. Then, we show the results of applying our proposal to thirteen case studies (small-scale real projects implemented in C) that were also used to validate the M2K methodology.

Keywords: measurement; object-oriented; paradigm; reengineering; high-level model; legacy software; design recovery.

I. INTRODUCTION

Building high-level models is a key discipline within the context of Software Engineering. Once the applications are designed and implemented, it is a common practice that the high-level models of existing applications are modified by designers in order to improve or upgrade their designs. When the result of the reverse engineering generates two models (an original and a modified one), it is interesting to detect the degree or percentage of modifications between them. This feature is relevant if both models keep the same structure by defining the same domain concepts despite the transformation. Object-oriented models may be mapped as graphs. Thus, it would be possible to apply isomorphism algorithms between two graphs in order to look for similarities. In our case, as the object-oriented models we are measuring have no relationships between classes, we map them as strings (an object-oriented element concatenation). Our proposal uses the *Levenshtein* distance [1] as a starting point and measures the distance between object-oriented models from a domain viewpoint.

The models are generated based on C source code using the M2K methodology [2].

This article is structured as follows: Section 2 summarizes the main concepts of M2K, which is a methodology that, by generating high-level models, allows us to understand the application structure of C source code. Section 3 summarizes the work related to our approach. Section 4 details the definition of the distance we introduce in this paper. Section 5 shows in detail how the distance is applied to a specific case study and the result of applying our proposal to a set of case studies. Section 6 concludes our paper.

II. M2K METHODOLOGY

M2K is a methodology that generates a high-level model from legacy C code [2]. It has two phases: *Source code Analysis* (supported by a tool termed *ModelMapper*) and *Expert mapping*.

The *Source code Analysis* is an automatic phase through which the code is parsed and mapped into a high-level model. The result of this phase is a group of *Class Candidates Definitions* (CCD), with no relationships between them. Each CCD represents a domain-specific concept and is implemented as a pair of elements (A, M), where A is a set of attributes and M is a set of methods.

The *Expert mapping* phase is manual and requires an expert who refactors the group of CCDs in order to improve the high-level model from a design viewpoint. In this paper we refer to the group of CCDs obtained by the automatic phase as the *Original Model* (OM), and to the refactored set of CCDs as the *Refactored Model* (RM). Our proposed distance is calculated between OM and RM.

III. RELATED WORK

In software engineering, the activity of measuring the software, or any part of its life cycle, is a well-known discipline [3][4]. There are several works that propose a metric of a given object-oriented (OO) model [5][6]. These ones measure a software artifact (or a process) and assign a meaning (e.g. coupling, complexity, cohesion) to what they

measure [7]. These metrics apply at the source-code and at the design levels [4]. On the other hand, the definition of thresholds for the majority of software metrics is a complex task [8]. *Xing et al.* [9] and *Lin et al.* [10] works also propose an approach to the identification of certain types of refactorings between two UML models. *Ohst et al.* [11] address the problem of how to detect and visualize differences between versions of UML documents. The most significant disadvantage of this approach is that the algorithms and tools are document type-specific. On the other hand, *Brun et al.* [12] propose a method -and a tool- where models are serialized into vectors of software entities to visualize the changes in the model.

Our proposal differs from other works in three main aspects: a) we focus on comparing two OO models, b) instead of applying a meaning to the measurement, we use the results to compare the models and infer their differences, and c) we focus on creating a measurement to know how much the M2K Methodology modifies the OM.

According to *Srinivasan and Devi*, "*the software metrics researchers proposing a new metric have the trouble of proof to show that the metric is adequate for measuring the software*" [13]. Thus, we consider that our approach is not a metric from a theoretical viewpoint. It fulfills the nine *Weyuker* properties [14] but the respective proof is a future work.

Several software applications from bioinformatics and automatic linguistic recognition systems require comparing long strings to find similar subsequences. Two similarity metrics frequently used by these applications are the *Hamming* and *Levenshstein* distances [15]. *Kolpakov et al.* propose a general viewpoint about the application of *Hamming* distance to computer science problems [16]. For example, *Kurtz et al.* apply this distance to find repetitive DNA structures [17]. *Norouzi et al.* propose a mathematical framework to find semantic similarities between images [18]. It evaluates the *Euclidean* [19] and the *Mahalanobis* distances [20] as a starting point of the proposal. The conclusion of this comparative analysis is the same as our proposal: the *Hamming* distance is a useful mathematical tool to measure similarities between entities. *Torralba et al.* propose a similar approach to the *Norouzi et al.* work but from an algorithmic viewpoint [21]. *Hamming* distance is defined in same-length words [22]. As in our proposal the measured models (or CCDs) usually have not the same length, we use *Levenshtein* distance that, in its definition, let measure two words with different length.

From our knowledge, there is no technique or methodology that compare two OO models using the *Levenshtein* distance in order to measure a transformation degree or difference between them.

The indicators designed in this approach to measure the transformation percentage between an OM and a RM is the main contribution of this paper.

IV. OUR PROPOSAL

In this section, we summarize briefly the *Levenshtein* distance definition, propose new measurements and explain how they are calculated. In this work, *Levenshtein* distance is applied to two types of words that: a model (OM and RM) or a CCD. It depends on the target software artifact.

A. Levenshtein Distance

The *Levenshtein* distance between two strings/words a, b (of length |a| and |b| , respectively) is given by $lev_{a,b}(|a|,|b|)$ where:

$$lev_{a,b}(\mathrm{i},\mathrm{j}) \begin{cases} \max(i,j) \quad if\ \min(i,j)=0, \\ min\begin{pmatrix} lev_{a,b}(\mathrm{i}-1,\mathrm{j})+1 \\ lev_{a,b}(\mathrm{i},\mathrm{j}-1)+1 \\ lev_{a,b}(\mathrm{i}-1,\mathrm{j}-1)+1_{(a_i \neq b_j)} \end{pmatrix} otherwise. \end{cases}$$

and $1_{(a_i \neq b_j)}$is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise.

This function may also be referred to as edit distance and indicates the minimum number of single-character edits (insertions, deletions and substitutions) required to change one word into the other.

For example, the *Levenshtein* distance between the word "**intentions**" and "**execution**" is 6, since the following six edits change one into the other, and there is no way to do it with fewer than six edits:

1. intentions → *ntentions (deletion of i)
2. ntentions → etentions (substitution of n with e)
3. etentions → exentions (substitution of t with x)
4. exentions → execntions (insertion of c)
5. executions → executions (substitution of n with u)
6. executions → execution (deletion of s).

This algorithm fits better to our proposal in comparison with the *Hamming* distance. For example, given two words with the same length "**flaw**" and "**lawn**", *Levenshtein* distance equals 2 (deletion of "**f**" from the front and insertion of "**n**" at the end), meanwhile the *Hamming* distance related to this example equal 4. Letters "**a**" and "**w**" are present in both words and the *Hamming* distance is not able to take this into consideration. In that way, we consider that *Levenshtein* distance performs better.

Software entities mapping applying *Hamming* distance. The *word* (that may be a model, a CCD, a set of attributes or a set of methods) is implemented as a vector. Each element of the original vector, that remains its concept after the refactoring process, will be in the same index of the refactored vector. In the same way, if an element of the original vector changes its concept from a domain viewpoint, it will be in a different index of the refactored vector. Accordingly, given two different words **A = (a_1, a_2, … , a_n)** and **B = (b_1, b_2, … , b_m)**, where $a_i \neq b_i$ for each i, **A'** and **B'** are vector implementations of **A** and **B**, **A' = <a_1, a_2, … , a_n, $null_1$, $null_2$,…, $null_m$>** and **B' = <$null_1$, $null_2$, … , $null_n$, b_1, b_2, … , b_m>**. For example, given the words **A = (C, A, N)** and **B = (M, E, N)** where **B** is the RM of **A**, the vector implementations of **A** is **A' = (C, A, N, $null_1$, $null_2$)** and the vector implementation of **B** is **B' = ($null_1$, $null_2$, N, M, E)**.

Concept **N** is in the same index position (3) of **A'** and **B'** because it represents the same concept in both models (the OM and the RM). In the others index position (1, 2, 4 and 5) **A' [i]** $\neq$ **B' [i]** because the *Refactoring phase* deleted the concept in the OM (index position 1 and 2) or added it in the RM (index position 4 and 5).

To simplify the reading of the paper when we refer to a *word*, we also refer to its vector implementation, indistinctly.

Equality between software entities. We consider that two elements (the original and the refactored) are equal when they represent the same concept from a design viewpoint. From an implementation viewpoint, the original element is equal to the refactored element when the refactored one was created by *Modelmapper* (the tool that supports the M2K methodology) in the *Source code analysis* phase, even when the *Expert mapping* phase removes arguments or refactor the name of the original element.

Maximum distance between software entities. To calculate our measurement, we define the maximum distance between software entities (a model or a CCD). The maximum distance is derived from the *Levenshtein* distance and supposes that both entities are completely different. In numerical terms, we consider this difference as a maximum distance.

Given two software entities **A** and **B** where **A** = (a_1, a_2, … , a_n) and **B** = (b_1, b_2, … , b_m), being $n \neq m$ and $a_i \neq b_i$ for each i, with i = min(n; m), then:

$$D_{max}(A, B) = \max(n, m)$$

We use this distance as a reference value. The measurement we propose is used to indicate how close or how far both models (or both CCDs) are from their maximum distance.

B. Distance definitions

Following, we enumerate our proposed definitions of indicators:

1. **Distance between models**: It is a distance between OM and RM. We can infer that if d(OM, RM) = 0, then they keep the same structure by defining the same domain concepts. If d(OM, RM) >> 0, we can infer that the transformation process modified significantly the OM from a design viewpoint.

2. **Distance between CCDs**: Given C_1 and C_2, where $C_1 \in$ OM and $C_2 \in$ RM are CCDs, C_1 and C_2 representing the same domain concept, we define d(C_1, C_2) as a number that indicates the transformation degree between C_1 and C_2. If d(C_1, C_2) = 0, we can infer that the transformation process did not modify the concept represented in C_1 in any of its aspects or features. In fact, this indicator is a list of distances that has as many elements as pairs of (C_1, C_2). Given an OM and a RM, there is a *Distance between CCDs* for each pair of CCDs (C_1, C_2).

3. **Difference between models**: It shows the difference in percentage between models, when compared to the potential maximum distance. Two identical models differ in 0%, while two completely different models differ in 100%.

4. **Difference between CCDs**: It shows the same indicator as the previous item but referring to two given CCDs. As it is a percentage related to the indicator *Distance between CCDs*, this indicator is a list.

To ease the analysis of the previous indicators, following we propose a global distance that includes and summarizes them.

5. **Global distance**: This indicator includes and integrates the previous indicators, and it is proposed in order to offer a general view. It is a weighted percentage between the *difference between models* and the *difference between CCDs*, considering the number of modified CCDs. The goal of this indicator is to summarize the previous ones.

C. Distance calculation

Once we have defined each indicator, we proceed to explain how to calculate them.

1. **Distance between models**: Let A be the OM and B, the RM, where model A = (a_1, a_2, … , a_n) and B = (b_1, b_2, … , b_m) are vectors of CCDs. We consider that $a_i = b_i$ when both CCDs represent the same concept. In this case $d_i(a_i, b_i) = 0$. Finally, by applying the distance defined in Section IV.A we obtain this indicator. Table I shows a small example where *Distance between models* is equal to 2.

TABLE I: Example of the distance between models.

	Model A	Model B	Distance
CCDs	CCD A_1	CCD A_2	0
	CCD B_1	CCD B_2	0
		CCD C_2	1
	CCD C_1		1
	d(A, B)		2

2. **Distance between CCDs**: Given C_1 and C_2, where $C_1 \in$ OM and $C_2 \in$ RM, representing C_1 and C_2 the same domain concept, we define four words to calculate d(C_1, C_2):

J: contains the vector of C_1 attributes.
K: contains the vector of C_1 methods.
M: contains the vector of C_2 attributes.
N: contains the vector of C_2 methods.

Thus:

$$d(C_1, C_2) = d_1(J, M) + d_2(K, N)$$

From a domain viewpoint, $J_i = M_i$ if both attributes represent the same concept, and $K_i = N_i$ if both methods represent the same functionality. This means that even if there can be syntactical differences between K_i and N_i, if both represents the same functionality, the distance is 0. Table II shows a simplified example where *Distance between CCDs* is 3.

TABLE II: Example of the distance between CCDs.

	CCD C_1	CCD C_2	Distance
CCDs	int a	int a	0
	char b	char b	0
	int c		1
	float d	<generic> d	0

		char e	1
Methods	int f1(a, b, c)	int f1(a, b, c)	0
	f2(d)	f2()	0
	char f3()	char f3(m, n)	0
		int f4(j, h)	1
	d(C_1, C_2)		3

It is worth explaining some important points related to Table II:

- In the transformation process, the **int c** attribute was removed from CCD **A** and the **char e** attribute was added in CCD **B**. We understand that both generate a distance since there is a domain modification in the software entity.
- We understand that the transformation of the **float d** attribute into **<generic> d** does not mean a domain modification. The new attribute **<generic> d** includes **float d** and as a result, it has not modified its domain meaning.
- The transformation process has carried out a syntactical modification in **f2()**, by removing its argument. In this case, as the functionality of **f2()** has not changed, even when the approach removes the arguments we consider that there is no distance. We apply the same criteria to **f3()**.
- The function **f4()** was created in CCD **B**, and as this behavior is not explicit in CCD **A**, a distance is generated.

3. **Difference between models**: Let A be the OM and B the RM, where model A = (a_1, a_2, … , a_n) and B = (b_1, b_2, … , b_m) are vectors of CCDs. We define it as follows:

$$Difference\ between\ models = \frac{d(A,B)}{D_{max}(A,B)} * 100\%$$

4. **Difference between models**: Let A be the OM and B the RM, where model A = (a_1, a_2, … , a_n) and B = (b_1, b_2, …, b_m) are vectors of CCDs. We define it as follows:

$$Difference\ between\ CCDs = \frac{\sum_{i=1}^{n} d(A_i,B_i)}{\sum_{i=1}^{n} D_{max}(A_i,B_i)} * 100\%$$

5. **Global distance**: Given OM = (COM_1, COM_2, …, COM_n) and RM = (CRM_1, CRM_2, …, CRM_m) *global distance* is defined as follows:

$$D(O_M,R_M) = \left(\frac{Matched\ CCDs}{D_{max}(O_M,R_M)} * \frac{d(O_M,R_M)}{D_{max}(O_M,R_M)} + \left(1 - \frac{Matched\ CCDs}{D_{max}(O_M,R_M)}\right) * \frac{\sum_{i=1}^{n} d(OM_i,RM_i)}{\sum_{i=1}^{n} D_{max}(OM_i,RM_i)}\right) * 100\%$$

where **Matched CCDs** is the number of CCDs of the OM that remains in the RM without modifications in its concepts from a domain viewpoint. Thus, it indicates the amount of times that the indicator *difference between CCDs* is applied.

V. VALIDATION

As this paper improves an existing approach, we use the same case studies presented in our previous work [2] to validate our proposal. In that work we have used as case studies thirteen small-scale real projects implemented in C. They come from two different sources: a) ten case studies were designed with UML classes and implemented by a group of advanced computer science students and b) three case studies were downloaded from different Internet websites.

A. Case studies

To give the reader an overview of the case studies, Table III shows the number of software entities of eight case studies we chose because they are the most representative in our analysis using M2K. We selected them to ease the comprehension and to show how the indicators work on different scenarios. Table IV shows the number of classes stated in the documentation/specification, the number of CCDs in the OM and the number of CCDs in the RM.

TABLE III: Number of software entities of each case study.

	Model A	Model B	Distance
Collection of CCDs	University.c		1
	Student_final	Student	0
	Student.h	Student_set	0
		BubbleSort	1
		Printing_Strategies	1
		Printing_Best_10	1
		Print_Avgs	1
		Print_Exams	1
		int f4(j, h)	1
	D(A, B)		6
	D_{max}(A, B)		10

TABLE IV: Number of classes in different analysis phases of each case study.

Case Study	LOC	Functions	ADT's	Vars	Modules
University	162	9	1	2	1
Calculator	200	9	0	0	0
Bank	216	5	7	2	0
MovieClub	116	3	3	0	1
AssemblyLine	168	8	2	0	2
BallotBoxes	228	17	4	1	4
LightBulbs	285	22	4	0	3
Elevators	295	19	4	1	4

TABLE V: Distance between Models for **University** case study.

Case Study	UML Classes	Initial CCDs	Refactored CCDs
University	1	3	7
Calculator	1	1	10
Bank	2	2	2
MovieClub	4	4	3
AssemblyLine	3	4	3
BallotBoxes	7	6	9
LightBulbs	6	4	8
Elevators	8	6	10

TABLE VI: Distance between **Student Final** and **Student**.

	Student_Final	Student	Distance
Atributes	student_code: int	student_code int	0
	grade: int	grade: int	0
Methods			0
	D(Student_Final, Student)		0
	D_{max}(Student_Final, Student)		4

TABLE VII: Distance between **Student.h** and **Student_set**.

	Student.h	Student_set	Distance
Atributes	grades: int	grades: int	0
		ELEMENTS: int	1
		STUDENTS: int	1
		finals: int	1
		averages: int	1
Methods	init_finals(…)	init_finals(…)	0
	search_exam_per_student(…)	search_exam_per_student(…)	0
	print_exam_per_student(…)		1
	average_per_student(…)	average_per_student(…)	0
	print_average_per_student(…)		1
	averages(…)	averages(…)	0
	BubbleSort(…)		1
	print_Best_10_averages(…)		1
	D(Student.h, Student_set)		8
	D_{max}(Student.h, Student_set)		18

In the following section, firstly we show in detail how our proposal is applied to one of the case study of the eight ones mentioned previously. Then, we show the indicators of the chosen eight case studies performing a brief analysis.

B. *Case study **University***

University is an application that takes information from two arrays in order to obtain a summary of the students information.

Distance between models. Table V shows the OM (Model A) of **University** generated by the automatic phase of M2K Methodology, and the RM (Model B) resulting from the *Expert mapping* phase. In this table we observe that the transformation process generated a *Distance between models* equals to 6. Since **University.c** no longer exists, the transformation process generated 5 extra CCDs to keep the same behavior of the model.

There are 2 CCDs (**Student_Final** and **Student.h** in Model A) where no transformations were applied. According to our proposal, they must be analyzed by using the indicator of Distance between CCDs (Tables VI and VII).

Distance between CCDs. Table VI shows the distance between the CCDs **Student_Final** and **Student.h**. We observe that the transformation process only generated a naming refactoring of each CCD. Although it is an improvement that eases the comprehension of the model, we do not consider that this refactoring implies a transformation from the domain viewpoint. Thus, the distance between these two CCDs is equal to 0.

Table VII shows the distance between the CCDs **Student.h** and **Student_set**. We observe a relevant variation from a design viewpoint. Four attributes were added in the CCD **Student_set** and four methods were deleted from the CCD **Student.h**. The transformation process corresponds to the distance between these two CCDs.

Difference between models. As $D_{max}(A, B) = 10$ (in Table V), the percentage of difference between models is equal to 60%.

Difference between CCDs. As the sum of all distances between CCDs is equal to 8 (8+0) and the sum of all maximum distances between CCDs is equal to 22 (18+4), the percentage of difference between CCDs is equal to 36%.

Global distance. As **Matched CCDs** is equal to 2, the *global distance* is equal to 55%.

We consider that the resulting indicators correspond to the transformations that the M2K Methodology generates from OM to RM. More specifically, the fact that **Global distance** is equal to 55% shows the relevant transformations between models, and between CCDs **Student_final** and **Student**.

C. *Global case studies analysis*

Table VIII shows the resulting indicators of the selected case studies. To ease the comparison between case studies, we include the resulting measured distances of the case study **University** (explained in the previous section).

Following some analysis related to Table VIII are suggested:

- **University**: We observe that even when the modified CCDs are high (80%), the refactorings in the classes were small (36%) but the impact of the analysis is given in the models (60%) in an intermediate value. Based on these indicators, we can infer that the OM has only been affected in a 50%.
- **Calculator**: In this case study, we observe that all values in the last four indicators are high. This situation is normal because the applied refactorings implemented a Command pattern to abstract the calculator operations, and the number of added classes was the same as the number of needed operators.
- **Bank**: Differently to the previous case study, this case study shows low values in its indicators, confirming that there is no difference between models. The **Global distance** measures the difference between CCDs by taking into account the percentage of modified CCDs. We consider that the result is coherent with what the expert analyzed for the test case.
- **MovieClub**: Conversely to the previous case study, our approach is applied on a case study where there is no *difference between CCDs* and there is a difference between models. Thus, the final indicator is composed only of the

TABLE VIII: Results of our approach applied to case studies.

	University	Calculator	Bank	MovieClub	AssemblyLine	BallotBoxes	LightBulbs	Elevators
D_{max}(OM, RM)	10	10	4	7	7	15	12	16
D(OM, RM)	6	8	0	1	1	5	4	4
$D(C_{OMi}, C_{RMi})$	8	7	6	0	4	11	7	16
$D_{max}(C_{OMi}, C_{RMi})$	22	9	30	0	26	51	65	77
Modified CCDs	80%	90%	50%	57%	57%	67%	67%	63%
Difference between CCDs	36%	78%	20%	0%	15%	22%	11%	21%
Difference between models	60%	80%	0%	14%	14%	33%	33%	25%
Global distance	55%	80%	10%	8%	15%	29%	26%	23%

difference between models by taking into account the percentage of modified CCDs.

- **AssemblyLine**: We consider that this case study is similar to **Bank** case study except for the *Difference between models* indicator. The final indicator shows the correspondence between both percentages in the differences (*Difference between CCDs* and *Difference between models*).
- **BallotBoxes**, **LightBulbs** and **Elevators**: These case studies are similar from the indicators viewpoint. We include them in order to show case studies with average results and to enforce the proposal validation.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a set of indicators that measure the transformation degree between two models. Although it is applied to the M2K methodology, we consider that it can be used for any pair of object-oriented models. Our approach offers four partial indicators and a global one. The *Global distance* includes and weighs two aspects, expressed in the partial indicators: the *distance between models* and the *distance between CCDs*. The percentages, that our proposal provides, measure the transformation degree between two models and ease the comprehension of the transformation itself. The indicators let us identify if the transformation process was applied to the model, to its members, or both. It is worth to remark that this proposal does not offer any interpretation (qualitative results) of the indicators (quantitative results). We consider that this proposal refines and empowers the M2K methodology by including a measurement methodology for its output. As future work, we will study the correspondence of these indicators with the set of refactorings defined in [2]. Some refactorings increase significantly our indicators and, conversely, others -despite they are repeatedly applied- do not.

Software engineers have agreed that there is a correspondence between the resulting indicators and the transformation degree between the OM and the RM, both from a design viewpoint. Although we evaluate the application in real case studies, we consider that it can be validated in larger applications. We also consider that the comparison between CCDs can be further refined by taking into account different distances between attributes and methods as two different weights. We will work to enforce the mathematical fundamentals of the approach in order to change the concept indicator by distance. Finally, we will prove that our approach fulfills the nine properties of *Weyuker* in order to be considered as a software metric [14].

REFERENCES

[1] Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady, 10:707.

[2] Cassol, I. and Arévalo, G. (2015). M2k an approach for an object-oriented model of c applications. In Evaluation of Novel Approaches to Software Engineering (ENASE), 2015 International Conference on, pages 250–256.

[3] Chidamber, S. R. and Kemerer, C. F. (1994). A metric suite for object-oriented design. IEEE Transactions on Software Engineering, 20(6):293–318.

[4] Mohagheghi, P. and Dehlen, V. (2009). Existing model metrics and relations to model quality. In Software Quality, 2009. WOSQ '09. ICSE Workshop on, pages 39–45.

[5] Lorenz, M. and Kidd, J. (1994). Object-oriented software metrics - a practical guide.

[6] Shatnawi, A., Seriai, A. D., Sahraoui, H., & Alshara, Z. (2017). Reverse engineering reusable software components from object-oriented APIs. *Journal of Systems and Software*, *131*, 442-460.

[7] Mohagheghi, P., Dehlen, V., and Neple, T. (2008). Towards a tool-supported quality model for model-driven engineering. In Proc. 3rd Workshop on Quality in Modelling (QiM'08) at MODELS, volume 2008, page 15.

[8] 8 Filó, T. G., Bigonha, M., & Ferreira, K. (2015). A catalogue of thresholds for object-oriented software metrics. *Proc. of the 1st SOFTENG*, 48-55.

[9] Xing, Z. and Stroulia, E. (2005). Umldiff: an algorithm for object-oriented design differencing. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 54–65. ACM.

[10] Lin, Y., Gray, J., and Jouault, F. (2007). Dsmdiff: a differentiation tool for domain-specific models. European Journal of Information Systems, 16(4):349–361.

[11] Ohst, D., Welle, M., and Kelter, U. (2003). Differences between versions of uml diagrams. In Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIG-SOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11, pages 227–236, New York, NY, USA. ACM.

[12] Brun, C. and Pierantonio, A. (2008). Model differences in the eclipse modeling framework. UPGRADE, The European Journal for the Informatics Professional, 9(2):29–34.

[13] Srinivasan, K. and Devi, T. (2014). A complete and comprehensive metrics suite for object-oriented design quality assessment. International Journal of Software Engineering and Its Applications, 8(2):173–188.

[14] Aggarwal, K., Singh, Y., Kaur, A., and Malhotra, R. (2007). Software design metrics for object-oriented software. Journal of Object Technology, 6(1):121–138.

[15] Todd, A., Nourian, M., & Becchi, M. (2017, December). A Memory-Efficient GPU Method for Hamming and Levenshtein Distance Similarity. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on* (pp. 408-418). IEEE.

[16] Kolpakov, R. and Kucherov, G. (2003). Finding approximate repetitions under hamming distance. Theoretical Computer Science, 303(1):135–156.

[17] Kurtz, S., Choudhuri, J. V., Ohlebusch, E., Schleiermacher, C., Stoye, J., and Giegerich, R. (2001). Reputer: the manifold applications of repeat analysis on a genomic scale. Nucleic Acids Research, 29(22):4633–4642.

[18] Norouzi, M., Fleet, D. J., and Salakhutdinov, R. R. (2012). Hamming distance metric learning. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, Advances in Neural Information Processing Systems 25, pages 1061–1069. Curran Associates, Inc.

[19] Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, pages 380–388. ACM.

[20] Jegou, H., Douze, M., and Schmid, C. (2011). Product quantization for nearest neighbor search. IEEE transactions on pattern analysis and machine intelligence, 33(1):117–128.

[21] Torralba, A., Fergus, R., and Weiss, Y. (2008). Small codes and large image databases for recognition. In Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on, pages 1–8. IEEE.

[22] Robinson, D. J. (2003). An introduction to abstract algebra. Walter de Gruyter.