Restricted metamodel-based similarity propagation: a comparative study

Gabriel Peschl and Marcos Didonet Del Fabro

FAES Lab. Federal University of Parana, Coronel Francisco H. dos Santos Street, Curitiba, Brazil

{gpeschl, marcos.ddf}@inf.ufpr.br

Abstract. In Model Driven Engineering (MDE), there are different approaches to establish links between elements of different models. The links are used for distinct purposes, such as serving as specification for model transformations. Once the links are established, it is common to set up a similarity value to indicate equivalence (or not) between the elements. The Similarity Flooding (SF) is one of the most know algorithms that may increase the similarity of elements that are structurally similar. The algorithm is generic, and it has proven to be efficient. However, it depends on a graph-based and generic encoding. In this paper, we present a comparative study of a couple of metamodel-based encodings. The goal was to verify if a less-generic implementation, involving a lesser number of model elements, based on the metamodel structures, could be a viable implementation and adaptation of the SF algorithm. We developed tests with two metamodels for managing bugs and their corresponding models: Mantis and Bugzilla.

Keywords. Matching model, constraint technique for propagation graph, Model-Driven Software Engineering, Similarity Flooding.

1 Introduction

In Model Driven Engineering (MDE), there are different approaches to establish links between elements of different models. The links are used for distinct purposes, such as, specification of model transformations, model traceability or data integration. The operation that establishes these links is called matching. The output of the matching comprises sets of mappings, or alignments, indicating how the elements relate to each other [4]. The alignments often have a value from 1 to 0, to indicate how close the linked elements are.

The Similarity Flooding algorithm [9] is a well-known algorithm that takes a set of initial similarity values and propagates them through structurally close elements [9]. The similarities 'flows' according to a propagation structure, that may vary depending on the encoding adopted. The use of adapted propagation algorithms may improve the similarities between the models elements [1][3][9].

Few authors implement the generic implementations for the Similarity Flooding that encode the model as a generic graph structure, such as the implementation in [9]. Most authors propose distinct ways to encode a (meta) model to perform the propagation similarities, as showed in [3].

In this article, we present a study of a set of propagation techniques, based on common specific structures of metamodels (e.g., attributes of references). We encode the propagation structures separately, aiming to verify if we can achieve good results using restricted implementation of the propagation graphs. We present a comparison of how much these similarities have been increased (or decrease) in comparison with an implementation comprising all the propagation structures, more similar (though not equivalent) to the original SF implementation. Our contribution is to verify if the development of constrained propagation techniques is advised, which could be tailored and applied in several MDE operations. We tested our studies with two partial metamodels: Mantis and Bugzilla.

This paper is organized as follows. Section 2 shows the context of this paper: MDE, matching and the Similarity Flooding algorithm. Section 2 also presents the propagations techniques, results and discussion, where we compare our propagation techniques with a generic method. Section 3 presents the related works. Section 4 presents the conclusions as well as the future work.

2 Context

In the MDE, models are first-class entities [1][4]. In this paper, a model represents a software, with notations and characteristics of interest [4]. A formal definition describes a model as a directed labeled multigraph [6]. Below, we provide the formalism regarding directed labeled multigraph and model (the definitions are based on [6]):

- A **directed labeled multigraph** $G = (N_G\ E_G, \Gamma_G)$ consists of sets of nodes N_G, sets of edges E_G and a mapping function $\Gamma_G : E_G, \rightarrow N_G \text{ X } N_G$.
- A **model** M is a triple (G, ω, μ), where G relates to a directed labeled multigraph; ω is itself a model called the reference model of M and, ω associates with a multigraph $G_\omega = (N, E_\omega, \Gamma_\omega)$. The expression $(\mu : N_G \cup E_G \rightarrow N_\omega)$ associates nodes and edges of G to nodes of G_ω.

An approach for MDE is based on metamodeling, which defines grammar and vocabulary to express models [6]. MDE approaches are often presented as a 3-level architecture: the 3rd level is called metametamodel; the 2nd describes the metamodel and the 1st has the terminal model [7]. Informally, a metamodel defines the valid model elements that may be instantiated. On its turn, a metametamodel defines the valid elements of a metamodel [7]. The terminal model represents a model itself [6]. The relation between these levels is called conformance [6] [4].

As already stated, we aim at producing better links between the (meta) models, through a process called matching. According to [9][1], matching is the process of establishing semantic correlations between model elements belonging to different

models. The output of matching is a set of links (alignments). Links may be created manually, but this process could be tedious in a large set of elements, or error prone. The solution is to use semi-automatic matching algorithms to automate these processes [9]. Link have similarities values in a range from 1 to 0 [4], which means *in common* or *not in common*. These similarities values could be calculated using an algorithm of Edit Distance, string equality, similarity, and others [4].

While the similarity values could be constant, we may improve them using the Similarity Flooding algorithm [9], which has been used in different scenarios and it has be proven to be efficient. The similarities may be used as an input to the Similarity Flooding algorithm, which are "propagated" along the links of the (meta) models. We explain a generic propagation of the Similarity Flooding [9] in the following.

Consider two input models M_a and M_b and their instances $\{a, a'\} \in M_a$ and $\{b, b'\} \in M_b$. Elements $\{a, a'\}$ as well as $\{b, b'\}$ are connected by a labeled-edge. The pairs $\{a, a'\}$ and $\{b, b'\}$ has similarities equaling x and y, respectively. The main idea of Similarity Flooding algorithm is to propagate the similarities between links in an iterative sum, which are connected by the same labelled-edge. The algorithm propagate x to (b, b') and update y [1]. The propagation coefficient, given as π, indicates how much the link similarity flows along the links of the (meta) models [9].

2.1 Methodology

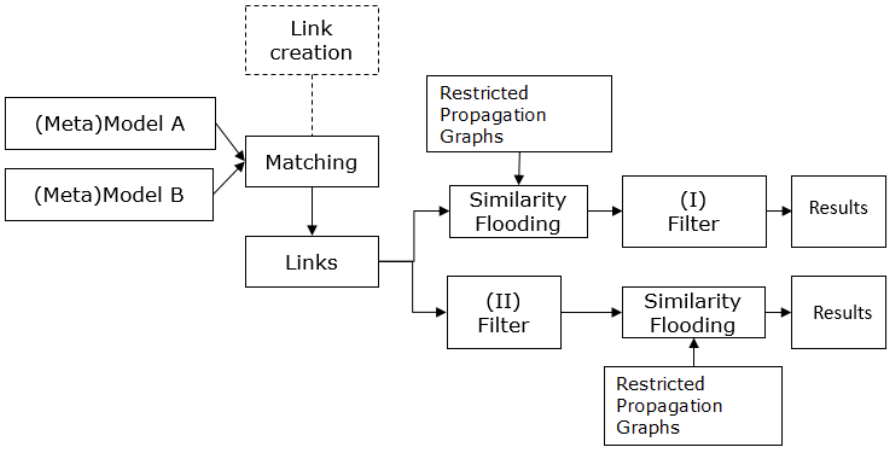

Fig. 1. Methodology proposed

In this section, we present how the similarities and the propagation are calculated. First, we load 2 models and theirs respective metamodels (Figure 1). After that, the execution of a matching algorithm creates the links between the model elements. We

use a well know string metric, called Levenshtein Edit Distance [8], to set up the similarities of the links. These are the initial input values for the Similarity Flooding algorithm.

We divided our experiments into 2 groups, according to a filter, which selects only links with a similarity higher than a given value. This enables diminishing the number of processed links.

- **Group I**: a filter is applied after running the Similarity Flooding algorithm;
- **Group II:** a filter is applied before running the Similarity Flooding algorithm.

The Similarity Flooding algorithm runs according to sets of propagation techniques constrained. We compared them with a generic solution at the end. The overall process is the following:

1. To perform the matching to create the links;
2. To calculate the similarities of these links using the Levenshtein Edit Distance [8];
3. To execute the Similarity Flooding Algorithm;
4. To run the filters:
   4.1. After executing the Similarity Flooding algorithm and,
   4.2. Before executing the Similarity Flooding algorithm;
5. To compare and discuss the results.

2.2 Existing methods

We encoded restrictive propagation graphs according to structural information of elements of a given (meta) model: class, attributes and references. We present the techniques in the following[1]:

– **From links between Classes to links between Attributes:** it propagates the similarity from link between classes to link between attribute belonging to the same matched class [1] (Figure 2). The propagation coefficient is calculated as: $\pi = 1/Lx$, where Lx indicates the number of links between attributes regarding *Class A* and *Class B* matched, with $Lx \neq 0$.

[1] Figures 2, 3, 4, 5 and 6 describe the propagation techniques. While they are quite similar, they illustrate the direction of the propagation between links. We use a partial representation of models of the Mantis (model A) and Bugzilla (model B) to explain the propagation techniques.

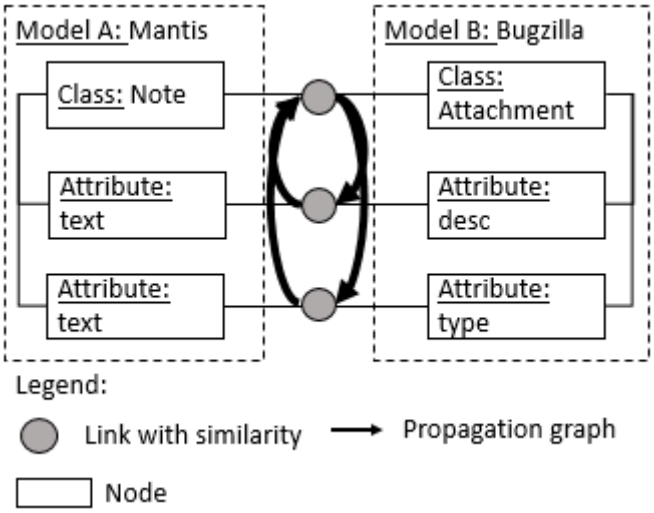

Fig. 2. Propagation technique: from links between classes to links between attributes

– **From links between Classes to links between References:** it propagates the similarities from links between classes to links between references belonging to the same matched class [1] (Figure 3). This is very similar to the previous propagation technique. The propagation coefficient is calculated as: $\pi = 1/Ly$, where Ly is the number of links between references of *Class A* and *Class B* matched, with $Ly \neq 0$.

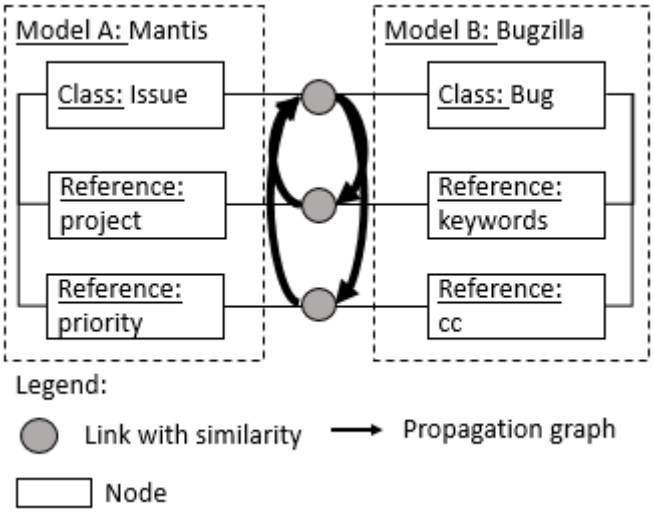

Fig. 3. Propagation technique: from link between classes to link between references

— **From links between Classes to link between Attributes and References:** it propagates the similarities from links between classes to links between attributes and references belonging to the matched class (Figure 4). The formula of the propagation coefficient is: $\pi = 1/Lxy$, where Lxy is the amount of the links between attributes and references of Class A and *Class B* matched, with $Lxy \neq 0$.

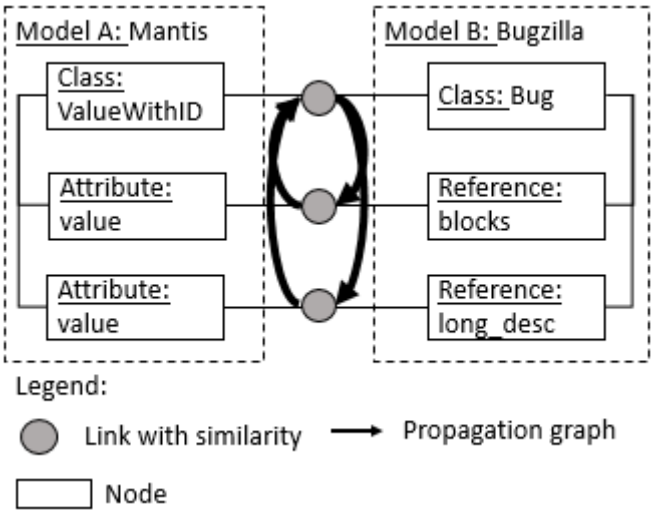

Fig. 4. Propagation technique: from links between classes to link between attributes and references

— **From links between Classes to link between References and Attributes:** it propagates the similarities from link between classes to links between references matched with attributes of the same class (Figure 5). It changes the propagation direction, if compared with the previous method. The propagation coefficient is $\pi = 1/Lyx$, where Lyx is the amount of links between references and attributes regarding *Class A* and *Class B* matched, with $Lyx \neq 0$.

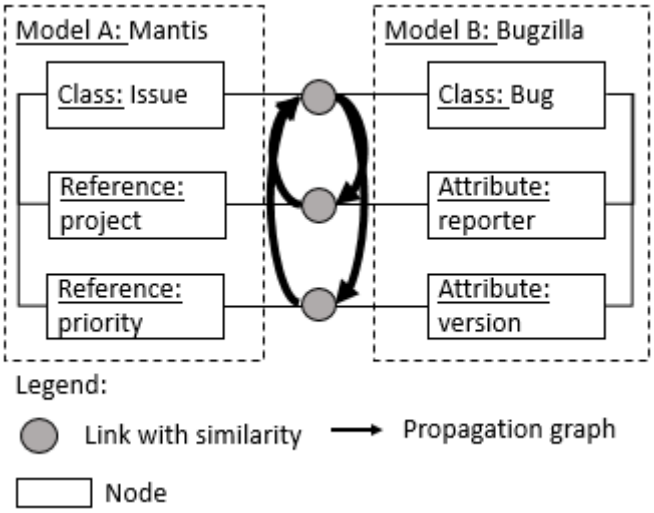

Fig. 5. Propagation technique: from link between classes to link between references and attributes

— **From links between Attributes to links between model instances:** the similarities between links of attributes are propagated to links of its respective instances [9] (Figure 6). The propagation coefficient is $\pi = 1/L_I$, where L_I designates the amount of instances of a given attributes regarding *Class A* and *Class B* matched, with $L_I \neq 0$.

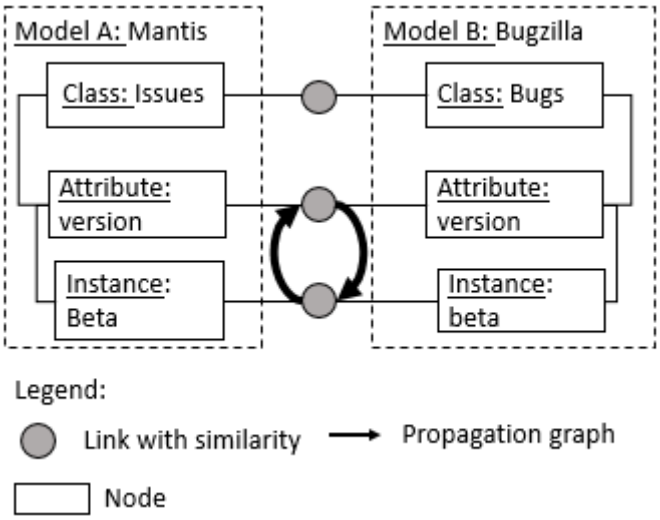

Fig. 6. Propagation technique: from links between attributes to links between model instances

2.3 Evaluation

We utilized the Eclipse Modeling Framework, EMF - a standard for MDE, to handle models and metamodel and encode the propagation techniques. We validate our proposal through two partial (meta) models: Mantis and Bugzilla[2], which are semantically similar, used to store information related to bug tracking projects [4].

The Mantis' metamodel has 9 classes, 15 attributes and 10 references. The Bugzilla's metamodel has 9 classes, 39 attributes and 8 references. The matching between these metamodels generated:

- Links between classes: 9 * 9 = 81;
- Links between attributes: 15 * 39 = 585;
- Links between references: 10 * 8 = 80.

While an instance is explicitly related to a class, we make a distinction between the model elements representing a class (called a class instance) and the model elements representing the values of the attributes (called an attribute instance). The Mantis´ model has 5 classes instances, 12 attribute with 1 attribute for each instance. The Bugzilla´s model has 4 classes instances and 31 attribute instances with 1 attribute per instance. The matching generated:

- Links between instances of classes: 5 * 4 = 20;
- Links between instances of attributes: 12 * 31 = 372;
- Links between instances: 12 * 31 = 372 (one attribute to each instance).

Table 1 shows the results according to each kind of propagation used. An approach combining the different methods is shown on Table 2. Both tables display the results of the (I) first filter and (II) second filter configurations; π indicates the propagation coefficient; the link with (*) represents the links between classes or links between attributes. The settings of iterations for the Similarity Flooding are 1, 3 and 6.

The value of the π differ for number of the links according to the filter configuration. For example, the link of the class "*IdentifiedElt x LongDesc"* has 4 links in the first filter configuration, then, $\pi = 1/L_x = ¼ = 0.25$. In other hand, the same link, in the second filter configuration, has 1 link to execute the propagation, then, $\pi = 1/L_x = 1/1 = 1$. This logic follows for all other links of model elements. Note that the propagations are restricted to a given type of elements, so they cannot be used in all propagation techniques.

[2] The metamodels of Mantis and Bugzilla are available on http://www.emn.fr/z-info/atlanmod/index.php/Ecore

Table 1. Partial results of constrained-based propagation techniques

Links	Sim.	π		1st		3rd		6th	
		I	II	I	II	I	II	I	II
Propagation: from links between Classes to links between Attributes									
*IdentifiedElt x LongDesc	0.091	0.25	1	0.047	0.195	0.047	0.195	0.047	0.195
id x who	0.25	0.25	1	0.018	0.195	0.013	0.195	0.012	0.195
* Issue x Bug	0.2	0.0075	1	1	0.686	1	0.686	1	0.686
version x version	1	0.0075	1	0.068	0.686	0.022	0.686	0.009	0.686
* Attachment x Attachment	1	0.0312	0.333	0.347	1	0.347	1	0.347	1
size x id	0.25	0.0312	0.333	0.019	0.333	0.019	0.333	0.019	0.333
Propagation: from links between Classes to links between References									
* Issue x Bug	0.2	0.014	1	1	1	1	1	1	1
attachments x attachment	0.5	0.014	1	0.064	1	0.028	1	0.015	1
Propagation: from links between Classes to links between References and Attributes									
* Issue x Bug	0.2	0.004	0.25	1	1	1	1	1	1
priority x priority	1	0.004	0.25	0.038	0.162	0.013	0.228	0.005	0.247
* Issue x BugzillaRoot	0.083	0.025	1	0.183	0.176	0.183	0.176	0.183	0.176
reporter x exporter	0.333	0.025	1	0.012	0.176	0.006	0.176	0.004	0.176
Propagation: from links between Classes to links between Attributes and References									
*IdentifiedElt x Bug	0.07	0.1428	1	0.243	1	0.243	1	0.243	1
id x cc	0.333	0.1428	1	0.079	1	0.046	1	0.036	1
Propagation: from links between attributes to links between model instances									
* version x version	1	1	-	1	-	1	-	1	-
Beta x beta	1	1	-	1	-	1	-	1	-
* category x exporter	0.111	1	-	0.185	-	0.185	-	0.185	
website x teste	0.25	1	-	0.185	-	0.185	-	0.185	-
* value x component	0.111	1	-	0.180	-	0.180	-	0.180	-
low x link	0.25	1	-	0.180	-	0.180	-	0.180	

Table 2. Partial results of the combined technique

Links	Sim.	π		1st		3rd		6th	
		I	II	I	II	I	II	I	II
*IdentifiedElt x LongDesc	0.091	0.25	1	0.013	0.070	0.013	0.070	0.014	0.070
id x who	0.25	0.25	1	0.005	0.070	0.003	0.070	0.003	0.070
* Issue x Bug	0.2	0.002	0.142	1	1	1	1	1	1
version x version	1	0.002	0.142	0.019	0.211	0.006	0.159	0.002	0.149
attachments x attachment	0.5	0.002	0.142	0.009	0.108	0.003	0.134	0.002	0.141
project x product	0.333	0.002	0.142	0.006	0.07	0.003	0.125	0.002	0.140
* Issue x BugzillaRoot	0.083	0.117	0.5	0.193	0.291	0.193	0.291	0.193	0.291
reporter x exporter	0.333	0.117	0.5	0.006	0.074	0.003	0.125	0.002	0.140
*IdentifiedElt x Bug	0.07	0.344	1	0.066	0.083	0.067	0.083	0.070	0.083
id x cc	0.333	0.334	1	0.006	0.083	0.003	0.083	0.002	0.083
* Attachment x Attachment	1	0.0312	0.333	0.097	0.125	0.091	0.125	0.081	0.125
size x id	0.25	0.0312	0.333	0.005	0.062	0.003	0.062	0.002	0.062

2.4 Discussion

We implemented separated propagation structures based on the metamodel elements to propagate similarities between links of model or metamodel elements. The techniques are rather simple to implement, but executing them separately enables to have some conclusions about the similarity process. We encoded five propagations and we executed them with two different filters. Table 3 give us an overview about the results of each propagation, compared with a combined approach. It shows the percent of gain over iteration of the Flooding algorithm and, the results according to (I) filter configuration and (II) filter configuration.

With respect to the links between the metamodels elements. The similarity propagation may be executed several times, until a given delta is no longer achieved. The Similarity Flooding algorithm tends to propagate the similarity to a class or element that has the biggest number of links. For these reason, after each iteration and normalization of the results, the similarity increased basically in a single kind of link.

Concerning the propagation between attribute links and instance links, we observed that the similarities are unable to "flow" between links, where the propagation coefficient is equal 1. For example, from the link *"version x version"* to the link *"Beta x beta"*, the same similarity of the link of the attribute is equal to the similarity of

the link of the instance, in any iteration. We can apply this logic where the propagation coefficient is equal 1 (Table 1).

The second filter configuration acted as a constraint, reducing the number of the links. Therefore, we did not find a significant increase relative to the average of the similarity in comparison with the combined technique.

As a limitation of this study, our implementation does not deal with links concerning inheritance propagation. However, it is possible to infer that abstract classes with a generic structure (e.g. an "Object" class), would have several links and the similarity would highly increase.

The way link were filtered is also a limitation. Del Fabro and Valduriez [1] show that such an implementation can be a disadvantage, since the limit value for the filter is known. In this work, we set the filter threshold according to the links between attributes or links between instances, with values equal to 0.5 for the links between attributes, and 0.25 for the links between instances. After empiric analysis, these values returned the links with a best match.

Table 3. Comparative results

	1st		3rd		6th	
Constraints	**I (%)**	**II (%)**	**I (%)**	**II (%)**	**I (%)**	**II (%)**
From links between Classes to links between attributes	84.9	172.6	83.6	161.9	86.3	164.4
From links between Classes to links between references	754.2	515.7	779.5	515.6	803.8	515.6
From links between Classes to links between references and attributes	169.5	81.1	175.1	81.1	182	81.1
From links between Classes to links between attributes and references	134.5	515.7	128.3	515.6	130	515.6
From links between attributes to links between model instances	427.2	-	462	-	483.7	-

3 Related works

Most of the existing approaches developing the SF algorithm and propagations, as well as its variants have similar goals: to produce alignments to integrate data or, to try to produce better alignment by exploiting elements in several formalism: xml files, ontology or metamodel [1][2][4][5][9].

Melnik [9] developed the Similarity Flooding algorithm to align data schemas and, nowadays, this structure may useful in many scenarios: metamodels alignments, ontology alignments and model alignments. In schema matching, we also have the Coma++ [2], that employ the Uppropagation. As opposed the Similarity Flooding algorithm, the Uppropagation propagates the mean of the highest similarity values from the instances to attributes [2].

Falleri et al [3] use the Similarity Flooding to find alignments in two metamodels in an automated way. The main contribution is the study on the various scenarios for encoding the metamodels into a directed labeled graph that can be exploited by the Similarity Flooding Algorithm [3]. In this work, we propose 5 restricted propagation techniques, based on metamodel structures. Heraguemi et al [5] propose the use of the Similarity Flooding algorithm to align software architecture metamodels [5]. Del Fabro et al [1] proposed a generic way to produce metamodels alignments using the Similarity Flooding algorithm in a chain metamodel transformation. The similarities are propagates on links between the metamodels elements and, these results are saved in a metamodel. Zhang, Yuan and Huan [12] developed a version of the SF for the MapReduce to be applied in a large-scale graph datasets. Here, each iterative sum of SF is a job of the MapReduce. According to results, the customized SF achieved its objective [12]. Truong et al [11] implemented a new version for SF in the context of integration of the ontologies. The method of the concept classification is applied to increase the precision and reduce the process of the SF. According to Truong et al [11], this method avoid the SF to compare exhaustively among all nodes of the ontology.

According to our knowledge, these are between the most representative approaches in its field of study. While they all have similarities, it is difficult to compare them, because the scenarios, the model encodings and the propagation techniques have differences on the conceptual design and implementation. This work gather some simple techniques and provide a comparison.

4 Conclusion

In this paper, we developed different propagation techniques based on metamodel structures, in order to execute a variant of the Similarity Flooding Algorithm [9]. The techniques may be handled by different existing approaches, though they are not executed separately in order to perform comparisons.

We implement propagations between five different kinds of elements, from metamodels or models. The main goal is to calculate the propagation coefficient between the existing links. The coefficient is calculated based on the structural information of a given (meta) model: class, attribute and reference. We presented a case study where we performed the algorithm in two models: Mantis and Bugzilla.

The restricted propagation executions had a high increase on the similarity values. Therefore, these techniques show viable, because they reduce the number of model elements. We argue that the greater the amount of incoming propagation graphs in a

single link, the higher its result. We can confirm as an example the link "Issue x Bug" (Table 1).

However, the metamodel and model instances are quite simple and they had concentrating classes. The combined technique would probably be more appropriate for highly and evenly connected model elements.

We also implement two filters on the Similarity Flooding algorithm results: (a) after executing the Similarity Flooding and, (b) before executing the Similarity Flooding. Both implementations enable to diminish the number of links that are involved in the propagation process. However, the choice of the filter value remains empiric. We can clearly see that knowing the models and metamodels in advance may have an influence on the choice of the propagation technique, especially to avoid the highly centralized elements. However, this should be tested and developed with bigger models and with different nature as well.

Future work should address the development and testing of the propagation techniques with larger models, and, testing the techniques with models of different natures.

5 References

1. Del Fabro, M. D., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Software & Systems Modeling, 8(3):305–324 (2009).
2. Engmann, D., Massmann, S.: Instance matching with coma++ (2007).
3. Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In Model Driven Engineering Languages and Systems, pages 326–340. Springer (2008).
4. Garces. K. Adaptation and evaluation of generic model matching strategies. PhD thesis, Université de Nantes (2010).
5. Heraguemi, K. E., Abid, A., Amirat, A.: Software architecture matching based on similarity flooding algorithm. Proc. 2nd International Symposium on Modelling and Implementation of Complex Systems (2012).
6. Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In Gorrieri, R. and Wehrheim, H., editors, Formal Methods for Open Object-Based. Distributed Systems, volume 4037 of Lecture Notes in Computer Science, pages 171–185. Springer Berlin Heidelberg (2006).
7. Kleiner, M., Didonet Del Fabro, M., Queiroz Santos, D.: Transformation as search. In Gorp, P., Ritter, T., and Rose, L., editors, Modelling Foundations and Applications, volume 7949 of Lecture Notes in Computer Science, pages 54–69. Springer Berlin Heidelberg (2013).
8. Levenshtein, I. V.: Binary codes capable of correcting deletions, insertions and reversals. In Soviet physics doklady, volume 10, pages 707 (1966).
9. Melnik, S.: Generic model management: concepts and algorithms, volume 2967. Springer (2004).
10. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, 152:125–142 (2006).

11. Truong, H. B., Nguyen, Q., U., Nguyen, N., T., Duong, T. H.: A new graph-based flooding matching method for ontology integration. Proc. 2013 IEEE International Conference on Cybernetics, pages 86-91 (2013).
12. Zhang, J., Chunfeng Y., Yihua H.: Parallelized Similarity Flooding Algorithm for Processing Large Scale Graph Datasets with MapReduce. Proc.13th IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies, pages 184-188 (2012).