AN APPROACH AND A TOOL FOR MERGING A SET OF MODELS IN PAIRWISE WAY

Mohammed Boubakir[1*] and Allaoua Chaoui[2]

[1,2]MISC Laboratory, Department of Computer Science and its Applications, Faculty of NTIC
University Constantine 2-Abdelhamid Mehri, Algeria

Email: boubakirmohamed@yahoo.fr[1*] (corresponding author), a_chaoui2001@yahoo.com[2]

DOI: https://doi.org/10.22452/mjcs.vol34no1.2

ABSTRACT

Model merging addresses the problem of combining information from a set of models into a single one. This task is considered to be an important step in various software engineering practices. When many (more than two) models need to be merged, the most practiced technique is to perform the merge in a pairwise way, without considering the order of merging. The problem with this technique is that the resulting quality is not guaranteed because it is influenced by such an order. In this paper, we propose a pairwise approach for model merging aiming to provide better results by taking into account the order of merging. This approach proposes to combine models in an iterative process until obtaining only one model. At each iteration, we first compare each pair of models in order to measure the similarity between them and to identify the correspondences between their component elements. This is performed using two heuristic-based operators respectively named compare and match. After that, we identify the most similar pairs of models and merge them using a proposed operator. We have implemented our approach as tool support called 3M and evaluated it on a set of case studies.

Keywords: ***Model merging, Model comparison, Maximum weighted matching, Compare, Match, Merge.***

1.0 INTRODUCTION

With the increased adoption of MDE (Model-Driven Engineering) [1] in industry, model merging is becoming one of the indispensable tasks in different software engineering activities. Indeed, models are frequently developed independently by eventually different teams. Therefore, it will be necessary for many situations to combine a set of models into a single one to get a unified perspective over them, to capture the relationships, and to understand the interactions between them. The combination aims also to carry out diverse types of analysis like synthesis, verification, and validation [2] [3]. Model merging is usually used to resolve conflicts that occur when integrating independently developed modules into a single project [4] [5], but it is also used in diverse other domains. For example, model merging is used to build a global ontology from a set of local ones in the ontology research field [6], and to merge conceptual database schemas in the database field [7]. In Software Product Line Engineering (SPLE) [8], model merging is considered as a fundamental step when a set of similar product variants need to be migrated into a model-based Software Product Line (SPL) [9]. In the literature, model merging is defined as the process of combining information from two or more models into a single one [4] [9], and it is commonly implemented through the three following operators: *compare*, *match* and *merge* [10] [11]. The first and the second operators are used to detect relationships between models, while the third one is used to combine them into a single model [12].

Numerous approaches for model merging have been proposed in the literature, most of which make assumptions about the types of models to be merged or focus on only a sub-problem of model merging like model comparison. For example, the authors in [13] and [14] concentrate on the comparison of UML models. In [2] and [12], the authors address the problem of matching and merging Statecharts models. Kpodjedo et al. [15] study the problem of detecting many-to-many matches in diagrams. Other existing work [16] [17] [18] propose solutions for merging EMF (Eclipse Modeling Framework) models [19], Rubin et al. [9] propose a solution for model merging which consists of simultaneously processing all input models. The vast majority of these approaches have focused only on merging a pair of models. However, several software development activities like, for example, model versioning [20] [21] [22] and Software Product Lines [10] [23] [24] [25] require combining many models. To do this, models are often merged in a pairwise way, i.e., the input models are merged progressively, by combining only two models at a time, without considering the order of merging. This solution does not provide any guarantee on the quality of results, which depends on the order of merging [9]. The merge quality is the ability to matching the elements of models that are more similar to each other. To the best of our knowledge, the approach described in [9] is one of the rare works that provides a solution to deal with many (more than two) models and takes into account the quality of merge. It focuses only on the matching problem and proposes to simultaneously treat all input models. This approach has limited applicability to models with a relatively simple representation. In the case of complex models, it becomes very difficult to process simultaneously the set of input models, especially with the presence of references between the elements of models.

In this context, we address in this paper the following question: How can we obtain the best quality results when merging a set of models in a pairwise way? Our main idea is to consider the order of merging the input models to obtain better results. This work extends and refines our previous work [26] by proposing a more refined implementation of the three operators seen above (*compare*, *match* and *merge*), and using a generic representation of models. In this representation, each model contains a set of elements, and each element contains other elements (its sub-elements) or values. To this end, we make the following contributions:

- An implementation of the three operators: *compare*, *match*, and *merge*. This allows for processing a pair of models. The first and the second operators are heuristic-based and allow to identify correspondences between the elements of a pair of models, and between the sub-elements of a pair of elements. We use two heuristics including typographic and linguistic similarities between the vocabularies of different models. As we aim to improve the quality of results, we try here to combine each element with the most similar to it using the Hungarian method [27]. The *merge* operator is not heuristic-based and uses the result of the two previous operators to obtain the merged model.
- An algorithm for processing a set of models. Our main idea to improve the quality of results is to combine, as much as possible, the most similar pairs of models. Thus, we propose to iteratively merge the set of input models by repeating the following steps: We firstly compare input models to assign a similarity degree for each pair of them. Then we combine a subset of these pairs such that the sum of their similarity degrees is maximal based on an implementation of the Edmond's algorithm [28].
- Implementation and evaluation. We implemented, and evaluated our approach by applying it on a set of UML models. This implementation is freely available as an open source project on *GitHub* platform[1].

The remainder of this paper is organized as follows: Section 2 presents a motivating example. Section 3 outlines some basic concepts of model merging and gives some preliminary definitions. Our approach for model merging is presented with more details in Section 4. Tool support for this approach is described in Section 5. In Section 6, we evaluate our approach and discuss its advantages and limitations. In Section 7, we present related work. Finally, in Section 8, we conclude the paper and give an outlook on our future work.

2.0 MOTIVATING EXAMPLE

To motivate our work, we use the simple example of UML models presented in Fig. 1(a). This example has been inspired by [9] [26]. It contains three models respectively denoted by M_1, M_2, and M_3. Each model contains a set of elements and each element contains a set of sub-elements, where the elements are classes and the sub-elements are their attributes. For example, M_1 contains one element, the *University* class, while M_2 contains two elements, the *School* and *Student* classes. The *University* class contains three sub-elements: the class name which has the value *University*, and the *name* and *address* attributes. To simplify the manipulation of these models, and as illustrated in Fig. 1(b), we assign a unique identifier to each element (The number in brackets in the bottom right of the element) and each sub-element (The number in square brackets before the sub-element name).

The first step toward merging a set of models is to compare them to measure the similarity between them. The comparison result between the three models M_1, M_2, and M_3, is shown in Fig. 1(b). Every two elements belonging to two different models are connected by a line labelled by the similarity degree between them. Similarly, every two models are connected by a dashed line labelled by the similarity degree between them. In Section 4, we present more details on how to calculate the similarity degree between a pair of elements and between a pair of models.

[1] The web-based Git version control repository hosting service, https://github.com

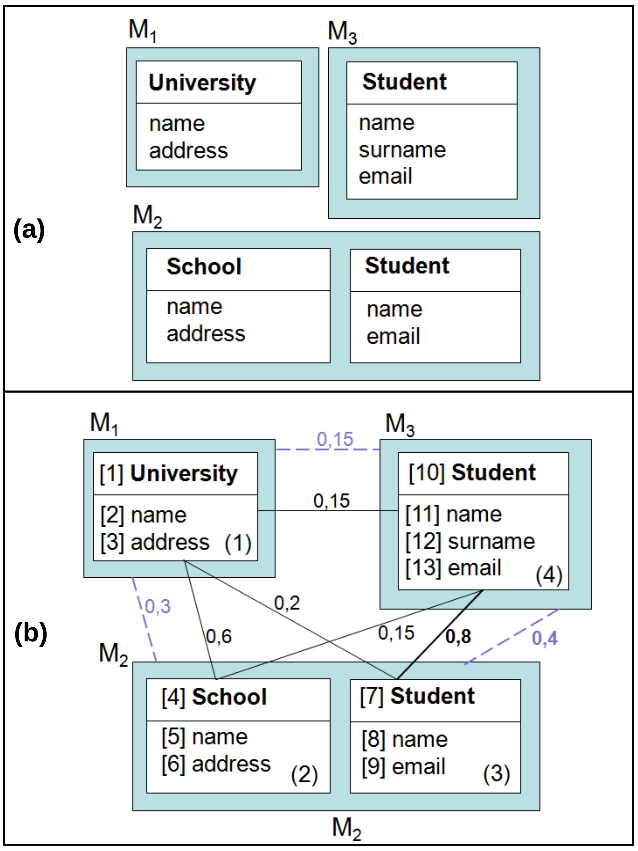

Fig. 1: Example of models and model comparison result: (a) Three simple UML models, and (b) the comparison result between the three UML models

As mentioned in [9] and in [26], there are different possibilities to merge a set of models and each of them may produce a result different from the others. For example, Fig. 2 presents the four possible models obtained by merging the three models of Fig. 1. These models are respectively denoted by *RM₁*, *RM₂*, *RM₃*, and *RM₄*. Each merge is distinguished from the others by how it combines the elements of the input models. For example, *RM₁* which contains two elements designated respectively (1, 2) and (3, 4) is obtained by combining element 1 with element 2, and element 3 with element 4, while *RM₂* is obtained by combining element 1 with element 3, and element 2 with element 4.

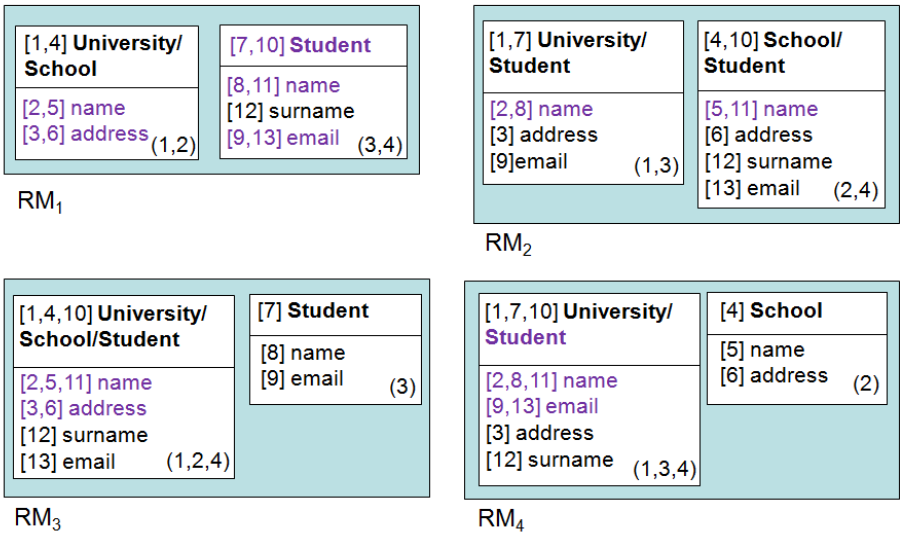

Fig. 2: Examples of merge results

The sub-elements that appear in blue in Fig. 2 are sub-elements shared by at least two original elements. Comparing the first and the second merges (The merge producing RM_1 and the one producing RM_2), it is obvious that the first one is better than the second one because it lets to combine the elements, which are the most similar to each other. More precisely, the first merge produces more shared sub-elements than the second one. The quality of a merge is measured using a weight such that the higher it is, the better the quality of the merge. We present in Section 3 how to assign a weight to a merge. Here are the weights of the merges that produce the four models of Fig. 2: 0.62, 0.16, 0.22, and 0.31. The first merge is best since it has the greatest weight.

In order to better understand the factors that influence the merge quality, we present below with details three possibilities to merge the three models of Fig. 1 in a pairwise way:

1. In the first step, we merge M_1 with M_2 by combining the elements 1 and 2. This produces a model with two elements (1, 2) and 3. After that, we merge the resulting model with M_3 by combining the elements 3 and 4 to obtain RM_1.
2. In the first step, we merge M_1 with M_2 by combining the elements 1 and 3. This produces a model with two elements (1, 3) and 2. Then, we merge the resulting model with M_3 by combining the elements 2 and 4 to obtain RM_2.
3. In the first step, we merge M_1 with M_3 by combining the elements 1 and 4. This produces a model with one element (1, 4). Finally, we merge the resulting model with M_2 by combining the elements (1, 4) with 3 to obtain RM_4.

As can be observed, the result and therefore the quality of a merge depend on two factors:

- The order of merging the input models: merging M_1 with M_2, then merging the result with M_3 is different from merging M_1 with M_3, then merging the result with M_2.
- The way in which the elements of a pair of models are matched: When we merge M_1 and M_2 by combining element 1 with element 2 does not give the same result as merging them by combining element 1 with element 3.

The approach presented in this paper aims to improve the merge quality by taking into account the two previous factors. Before presenting our approach in Section 4, the next section discusses model merging and the related concepts.

3.0 BACKGROUND AND PRELIMINARY DEFINITIONS

We present in this section some basic concepts of model merging and some preliminary definitions, which are necessary for the comprehension of our approach. First, we define the concept of model merging in which a model is assumed to be represented as a set of elements. Next, we briefly present different categories of model merging approaches. Then, we present the concept of a tuple and show how to assign a similarity degree to a given tuple. This concept is involved when merging many (more than two) models in addition to the concept of a pair of elements manipulated when merging a pair of models. After that, we present the concept of tuple match which is a way of structuring the elements of a set of models in tuples. Finally, we describe how to measure the quality of a merge.

3.1 Background on Model Merging

3.1.1 Definition

Model merging is a technique that aims to combine a set of models to provide a single model [4] [9]. It is defined as a process that involves three steps respectively performed using the following three operators: *compare*, *match*, and *merge* [9] [10] [11].

- The *compare* operator compares a pair of models to calculate the similarity between their elements. As result, it assigns for each pair of elements a number between 0 and 1 representing the similarity degree between them.
- The *match* operator allows deciding whether an element is the same as another one or not based on the result of the *compare* operator.
- The *merge* operator allows combining the elements of a pair of models to obtain a single model based on the result of the *match* operator.

3.1.2 Classification of Model Merging Approaches

Model merging approaches are classified into two and three-way approaches. They are also classified into state-based and change-based approaches.

Two-Way or Three-Way Merging: Two-way merging approaches merge two versions of a model regardless of their ancestor. By cons, three-way merging approaches perform the merge by exploiting the information of the common ancestor [22].

State-Based, Change-Based, or Operation-Based Merging: State-based merging approaches combine the elements of two model versions using only the information about the states of the models. By cons, change-based merging approaches exploit also the information about changes that have been made during the evolution of the model. Operation-base merging is a subcategory of change-based merging, in which changes are modeled in the form of operations [22] [29].

The approach presented in this paper is a two-way and a state-based approach, because the information about the common ancestors of models and the changes that brought models into their current states are not considered.

3.2 Preliminary Definitions

3.2.1 Tuple of Elements

Let $M = \{M_1, M_2, \ldots, M_n\}$ be a set of models and let E_M be the set of all elements of M which is defined as follows:

$$E_M = \cup_{i=1..n} E_i \qquad (1)$$

where E_i represents all elements of the model M_i. A tuple [9] of elements (or tuple for simplicity) denoted by t is defined as a non empty subset of E_M. If a tuple does not contain two elements from the same model, it is said to be valid. Formally:

$$\forall (e_1, e_2) \in t^2, \forall (M_1, M_2) \in M^2, (e_1, e_2) \in M_1 \times M_2 \Rightarrow M_1 \neq M_2 \qquad (2)$$

The two sets of elements {1, 2}, and {1, 2, 4} of Fig. 1 represent two examples of valid tuples respectively denoted by (1, 2) and (1, 2, 4). However, the set of elements {1, 2, 3} is not a valid tuple because it contains two elements, 2 and 3, which belong to the same model M_2.

3.2.2 Similarity Degree of Tuples

A similarity degree (a value between 0 and 1) is assigned to each tuple using the formula 3, which assumes that each model is represented as a set of elements, and each model element contains a set of sub-elements. Furthermore, it assumes that the relationship between a sub-element and its parent element is indicated by a role. For example, in the models illustrated in Fig. 1, the role *Class name* indicates the relationship between the sub-element 1 and the element 1, while the role *Class attribute* indicates the relationship between the sub-element 2 and the element 1. A criterion is defined for each role and a weight is assigned to each criterion. Table 1 gives an example of roles and their corresponding criteria for the UML class diagram. Formula 3 assigns the zero value to each invalid tuple and to each tuple containing a single element.

$$\mathrm{S}(t) = \sum_{\mathrm{C}} \mathrm{w_c} \times \mathrm{S_c}(\mathrm{t}) \qquad (3)$$

S(t) is the function that calculates the similarity degree of the tuple t, w_c is the weight of the criterion c, and C is the set of criteria defined for the elements of the model where $\sum_{\mathrm{C}} \mathrm{w_c} = 1$.

Table 1: Example of roles and criteria

Role	Criterion	Weight
Class name	Similarity of class names	0.4
Class attribute	Similarity of class attributes	0.6

$S_c(t)$ represents the function that calculates the similarity degree of the tuple t considering only the sub-elements that correspond to the criterion c. The value of $S_c(t)$ is calculated using the following formula which has been inspired by the work of Rubin et al. [9]:

$$S_c(t) = (\sum_{2 \leq j \leq m} j^2 \times d_j)/(n^2 \times |\pi(t)|) \qquad (4)$$

where *m* is the size (number of elements) of *t*, *n* is the number of models, d_j is the distribution of sub-elements (the number of sub-elements that appear in *j* elements of *t*), and π (t) represents the set of distinct sub-elements of *t*. Let us, as an example, calculate the similarity degree of the tuple (1, 3, 4) of Fig. 1 based on the criteria of Table 1, we have: *m* = 3 (we have 3 elements in the tuple which are 1, 3, and 4), *n* = 3 (there are 3 models). For the criterion *similarity of class attributes*, both d_2 and d_3 are equal to *1*, because there is one sub-element (*email*) that appears twice, and one sub-element *(name)* that appears thrice in the tuple, and $|\pi$ (1, 3, 4)$| = |\{name, address, email, surname\}| =$ 4. As a result, we have: $S_{similarity\_of\_class\_attributes}$ = 0.36. In the same way, we calculate the similarity degree for the criterion *similarity of class names,* which is equal to 0.22. Finally, using the criteria weights of Table 1 and the formula 3**,** we obtain the similarity degree of the tuple (1, 3, 4), which is equal to 0.31.

3.2.3 Tuple Match

Let *M* be a set of models and let E_M be the set of all its elements defined using the formula 1, a tuple match (or a match for simplicity) denoted by *H* is defined on *M* as any set of tuples that satisfies the following properties [9]:

(a) Each tuple of *H* contains a subset of E_M.
(b) The tuples of *H* are all valid.
(c) The tuples of *H* are all disjoint, i.e., $\forall (t_1, t_2) \in H^2, t_1 \cap t_2 = \emptyset$.
(d) *H* is maximal, i.e., the two previous properties (*b*) and (*c*) will be violated, if we add any additional tuple to *H*.

For example, the two tuples (1, 2) and (3, 4) of Fig. 1 represent a tuple match denoted by ((1, 2), (3, 4)). But the tuple (1, 4) does not represent a tuple match because we can increase it by the tuple (2) or (3).

A similarity degree is assigned to *H* using the following formula [9]:

$$\mathrm{S_h}(H) = \sum_{\mathrm{t \in H}} S(\mathrm{t}) \qquad (5)$$

where *t* represents a tuple, and *S* is the function that calculates the similarity degree of tuples (see formula 3).

The four possible tuple matches of the models of Fig. 1 and their similarity degrees are presented in Table 2.

Table 2: Example of matches and their similarity degrees

Match number	1	2	3	4
Match	((1,2), (3,4))	((1,3), (2,4))	((1,2,4), (3))	((1,3,4), (2))
Similarity degree	0.62	0.16	0.22	0.31

Each tuple match represents a possibility of merging the set of input models to obtain a single one (each tuple match corresponds to a resulting model). For example, the four models of Fig. 2 are obtained by merging the three models of Fig. 1 according to, respectively, the four tuple matches of Table 2 (the *match* number *i* in Table 2 corresponds to the model RM_i in Fig. 2).

3.2.4 Quality of Merge

Similarly to [9], we measure the quality of a merge using a weight, which is equal to the similarity degree of the corresponding tuple match. The higher the weight value of a merge, the more it is considered a good merge. For example, the weight value of the merge producing the model RM_1 is equal to 0.62. It is the best merge because it has the largest value of weight. Measuring the merge quality using a weight will be useful to evaluate our proposed approach in Section 6.

4.0 OUR APPROACH

In this section, we present our pairwise approach for merging a collection of similar models. This approach aims mainly to enhance the resulting quality by considering the order of merging the input models, and the way of matching the elements of each pair of models. This is achieved by trying as much as possible to combine each model with the one that is most similar to it, and each element with the element that is most similar to it. The input models are merged using an iterative process, which is repeated until obtaining a single model, and which includes two basic steps: First, we compare each pair of models using our implementation of the *compare* and *match* operators. This allows measuring the similarity between every two models and identifying the best matching between their elements. In the second step, we identify the most similar pairs of models and then use our implementation of the *merge* operator to merge each of them.

In the rest of this section, before detailing our model merging algorithm (the algorithm that implements our approach), we first describe how we represent models. Then, we present our implementation of the three operators: *compare*, *match*, and *merge*, which are used by the model merging algorithm to handle each pair of models.

4.1 Model Representation

We present in this section our representation of models, which has been inspired by the one used in [10] and XMI (XML Metadata Interchange) principles [30]. We define a model as a tree, where the root corresponds to the model, and the rest of the nodes correspond to the elements of the model. Each edge in the tree corresponds to a composite relationship either between the model and one of its first-level elements or between an element and one of its sub-elements. Each element has a unique identifier, a type, and a role indicating the type of relationship that connects it to its parent. As an example, Fig. 4 shows a partial representation of the simple UML model presented in Fig. 3.

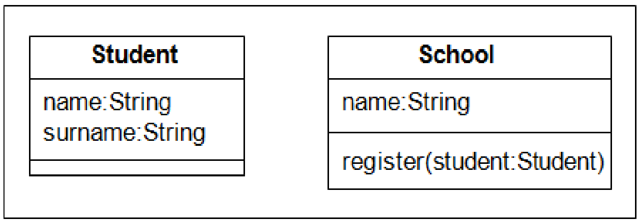

Fig. 3: A simple UML model

The types and the roles are defined depending on the type of the model. For example, for the UML class diagram, the set of types includes *Class*, *Attribute*, *Operation*, *Parameter*, *String*, *Reference*, etc. The set of roles includes: *Class*, *Class name*, *Class attribute*, *Class operation*, *Attribute name*, *Attribute type*, *Operation name*, *Parameter*, *Parameter name*, *Parameter type*, etc.

The element whose type has no owned properties, like for example, *String* or *Reference* is called *atomic*. The atomic elements represent the leaves of the tree, and each of them has its own value. The elements of other types, like for example, *Class* or *Attribute*, are called *compound* elements. The values of compound elements are obtained based on the values of their sub-elements.

Thanks to the type *Reference*, an element can play several roles at the same time. For this, the element of reference type carries the identifier of the referenced element and thus enabling it to play its role. For example, in Fig. 4, element 18[2] which is of reference type has a value equal to the identifier of element 1. Since it plays the parameter type role of element 16, element 1 plays also this role in addition to the class role.

To make it possible to represent the result of the merge, it is necessary for an element to have multiple attributes with the same role. For example, a class or an attribute in a UML class diagram can have more than one name. For this, we represent each element attribute as an element with its own identifier.

In order to apply the same treatment to different elements with similar characteristics, and consequently make model representation more generic, the elements are classified into the following categories:

- *Unordered-compound* elements: includes compound elements that are connected to their parents in any order, such as classes, class attributes, and class operations.
- *Ordered-compound* elements: includes compound elements that are connected to their parents according to an order, such as java method parameters. The elements of this category are represented by empty circles and each of them contains an attribute indicating its order. For instance, element 16 in Fig. 4 illustrates this category.

[2] The element whose identifier is equal to 18

— *Same-attribute* elements: includes atomic elements that represent the same attribute in the parent element but have different values, such as different names of a class, an attribute, or an operation in a UML class diagram. The elements of this category are represented by filled circles (see Fig. 4).

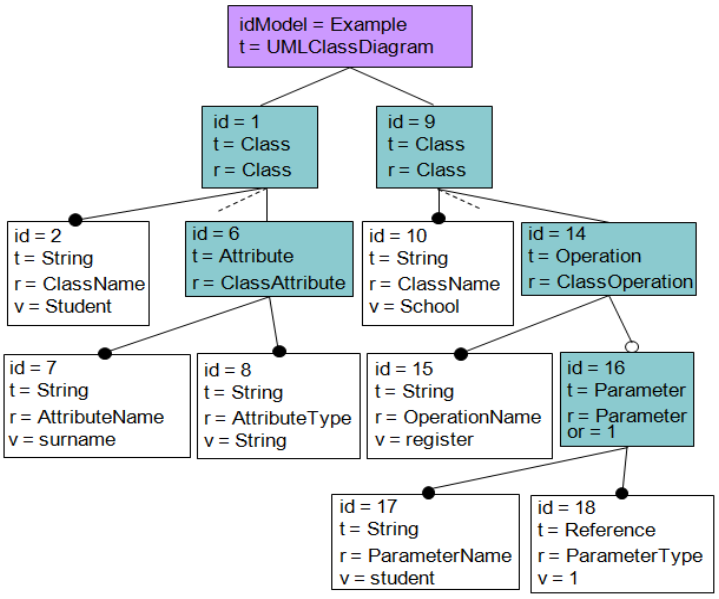

Fig. 4: Example of our model representation

4.2 Implementation of *Compare* and *Match* Operators

This section presents how to implement the *compare* and *match* operators. These two operators are applied to a pair of models in order to measure the similarity between them and to identify the best matching between their elements. A similarity function assigns a similarity degree (a number between 0 and 1) to each pair of models based on a set of user-defined criteria. The user defines a criterion and its weight (a value between 0 and 1) for each element role. Table 3 presents an example of criteria for a UML class diagram. The similarity function is defined by the following formula:

$$S_M(M_1, M_2) = \sum_{c\epsilon C} w_c \times compareAndMatch_c(E_1, E_2) \qquad (6)$$

where M_1 and M_2 represent the two models to be compared, C represents the set of criteria, and w_c represents the weight of the criterion c, where $\sum_{c\epsilon C} w_c = 1$. E_i represents the set of all elements whose role corresponds to the criterion c in the model M_i. For example, according to Table 3, the similarity between a pair of UML class diagrams is calculated using only the criterion *similarity of classes* as follows: $S_M(M_1, M_2) = compareAndMatch_{similarity\_of\_classes}(C_1, C_2)$ where C_1 and C_2 contain respectively all classes of M_1 and M_2. The next section gives more details about the function *compareAndMatch*$_c$.

Table 3: Example of criteria for comparing UML class diagrams

Element role	Criterion	Weight
Class	Similarity of classes	1

4.3 Compare and Match Function

The *compareAndMatch*$_c$ function accepts as input two sets of elements (respectively denoted by E_1 and E_2) having the same role. It identifies the matched pairs of elements and measures the similarity degree (a value between 0 and 1) between E_1 and E_2. For this, we firstly compare the elements of E_1 and E_2 in a pairwise way. Then, we identify the matched pairs of elements. A pair of elements is considered to be matched if the first element and the second one are considered to be the same. In our work, an element cannot be matched to more than one element. If an element does not match any other element, it is said to be unmatched. Finally, we calculate the similarity degree between E_1 and E_2.

In the following, we describe how to manipulate the elements of each of the three categories discussed above.

4.3.1 Unordered-Compound Elements

In the first step, we compare each pair of elements denoted by (e_1, e_2), where e_1 and e_2 belong respectively to E_1 and E_2 (E_1 and E_2 are the two input sets of elements). This is done by using a similarity function denoted S_e in a way similar to the one used above with a pair of models. This function assigns a similarity degree (a number between 0 and 1) to the pair (e_1, e_2) based on a set of criteria defined for each element role such that a criterion is attributed to each sub-element role. This set of criteria describes how to calculate the similarity degree between two elements having the same role. An example of criteria for comparing UML classes is presented in Table 9. The similarity degree between e_1 and e_2 is calculated using the following formula:

$$S_e(e_1, e_2) = \sum_{c \in C} w_c \times compareAndMatch_c(S_{\mathrm{E1}}, S_{\mathrm{E2}}) \qquad (7)$$

where C represents the set of criteria defined for the element role of e_1 and e_2, w_c represents the weight of the criterion c, S_{Ei} represents the set of all sub-elements whose role corresponds to the criterion c in the element e_i. For example, according to the criteria presented in Table 9, the similarity degree between two classes is calculated based on three criteria as follows: $S_e(e_1, e_2) = 0.5\times$ $compareAndMatch_{cn}(N_1, N_2) + 0.25 \times compareAndMatch_{ca}(A_1, A_2) + 0.25 \times compareAndMatch_{co}(O_1, O_2)$, where *cn*, *ca* and *co* represent respectively the criterion of class names, attributes and operations, N_1 and N_2 contain respectively class names of e_1 and e_2, A_1 and A_2 contain respectively all attributes of e_1 and e_2, and finally O_1 and O_2 contain respectively all operations of e_1 and e_2.

Once the comparison is completed, we combine two solutions to identify the matched pairs of elements. The first solution which is a threshold-based solution (e.g., [12] [13]) considers that two compared elements are similar if the similarity degree between them is greater than a threshold specified for the element role. The problem with this solution is that an element can be related to several other elements by similarity relations with values above the threshold. To overcome this problem, we use the second solution that is based on bipartite graph matching (e.g., [26] [31]). The matched pairs of elements are identified as follows:

1) We firstly replace each similarity degree that is below the threshold by zero. After that, we represent the comparison result as a bipartite weighted graph, where the two sets of vertices contain respectively the elements of E_1 and those of E_2. The edges between the elements are labeled by the similarity degree between them. The absence of an edge between two elements means that the similarity degree between them is equal to zero. An example of the comparison result between the elements of two sets respectively denoted by A and B is shown in Fig. 5. The dashed line connecting A and B is labeled by the similarity degree between them. All possible matchings produced when comparing A and B, and their similarity degrees are presented in Table 4.

2) As our approach aims to achieve a better quality result, we try as much as possible to match each element with the element that is the most similar to it. For this, we identify the best matching, i.e., the matching with the largest value of similarity degree. For example, matching number 3 in Table 4 is the best matching. We modeled this problem as the maximum weighted matching in bipartite graphs and resolved it in polynomial time with the Hungarian method [27].

Finally, we use the following formula to calculate the similarity degree between E_1 and E_2:

$$S(E_1, E_2) = \sum_{(e_i, e_j) \in M_{best}} S_e(e_i, e_j) / (|E_1| + |E_2| - |M_{best}|) \qquad (8)$$

where $S_e(e_i, e_j)$ returns the similarity degree between the two elements e_i and e_j according to the formula 7, M_{best} and $|M_{best}|$ represent respectively the best matching and the number of pairs it contains.

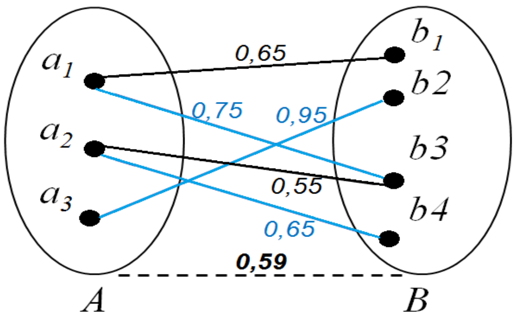

Fig. 5: An example of the comparison result between two sets of elements

Table 4: An example of possible matchings obtained when comparing two sets of elements

Number	1	2	3
Matching	(a_1, b_1), (a_2, b_3), (a_3, b_2)	(a_1, b_1), (a_2, b_4), (a_3, b_2)	(a_1, b_3), (a_2, b_4), (a_3, b_2)
Similarity degree	2.15	2.25	2.35

4.3.2 Ordered-Compound Elements

We manipulate the elements of this category as follows:

- If the two input set of elements E_1 and E_2 contain the same number of elements, we compare each element from E_1 with the element that has the same order in E_2. We calculate the similarity degree between two elements in the same way as with the first category. The similarity degree is equal to zero if the compared elements do not have the same order. A threshold is used to decide whether the two elements are similar or not. If all elements of E_1 have similar elements in E_2, the set of matched pairs contains each pair denoted (e_i, e_j) where e_i and e_j are similar, and the similarity degree between E_1 and E_2 is calculated as follows:

$$S(E_1, E_2) = \sum_{e_i \in E_1, e_j \in E_2} S_e(e_i, e_j)/|E_1| \qquad (9)$$

- If E_1 and E_2 have different number of elements or there is at least one unmatched element, the set of matched pairs of elements is an empty set and the similarity degree between E_1 and E_2 is equal to zero.

4.3.3 Same-Attribute Elements

First, we compare each pair of elements denoted (e_i, e_j), where e_i and e_j belong respectively to E_1 and E_2 (E_1 and E_2 are the two input sets of elements). Since the elements of this category are atomic elements of string type, like for example class names and attribute names, the similarity degree between e_i and e_j is directly calculated from their values using both linguistic and typographic matching. The linguistic matching is based on the N-gram algorithm [32]. This algorithm calculates the similarity degree between a pair of strings based on counting the number of their identical substrings of length *N*. The typographic matching calculates the similarity degree between a pair of strings based on their linguistic correlations. For this, it uses the *WordNet::Similarity* package [33]. We obtain the overall similarity degree by taking the maximum of the results obtained by the typographic and the linguistic matching.

Since the elements of E_1 (and respectively those of E_2) represent the same element, it is not necessary to find matched pairs of elements. We need only to calculate the similarity degree between E_1 and E_2. For this, we use the following formula: $max(S_e(e_i, e_j))$, where S_e returns the similarity degree between e_i and e_j.

4.4 Implementation of the Merge Operator

The *merge* operator uses the result of the *compare* and *match* operators to combine a pair of models in a single model. The elements of the merged model are constructed based on the elements of the two input models from top to bottom, i.e., we start with the first-level elements such as classes, and then those of the next one such as class names, and attributes, and so on. We manipulate the elements of the first and second category as follows:

- A new element denoted e_{1-2} is created by combining each matched pair of elements denoted by (e_1, e_2) such that:
  - e_{1-2} has the same type, the same role, and the same order (if e_1 and e_2 belong to the second category) as e_1 and e_2.
  - if e_1 and e_2 are first-level elements, e_{1-2} becomes a first-level element in the merged model; otherwise, it becomes a child of the element obtained by combining the parents of e_1 and e_2.

- All unmatched elements are directly copied into the merged model. If an unmatched element is a first-level element, it becomes a first-level element in the merged model; otherwise, it becomes a child of the element obtained from its parent.

- If two elements, respectively denoted by e_1 and e_2, are combined to obtain e_{1-2}:

- All their sub-elements which belong to the third category are directly copied into the merged model without redundancy, i.e., for each pair of identical (have the same value and the same role) sub-elements we create only one element. These new elements become children of e_{1-2}.
- Each two sub-elements which have the type *Reference* and which have the same role are combined if their referenced elements are combined. The resulting element has a value equal to the identifier of the element obtained by combining the two referenced elements.

4.5 Model Merging Algorithm

Based on the implementation of the three operators presented above, we propose in this section an algorithm (see *Algorithm 1* below) implementing our model merging approach [26]. This algorithm takes as input a set of models (denoted by *M*), and merges them progressively in a pairwise way by repeating three basic steps until merging all input models (until *M* contains a single model). The algorithm uses three variables denoted by *T*, *P*, and *m*. T is a set of 3-uplets, where the first and the second term of each 3-uplet represent respectively two models, while the third term represents the similarity degree between them. *P* is a set where each element represents a pair of models. Finally *m* represents a model.

Algorithm 1. Our model merging algorithm

```
1:  input: M
2:  var: T, P, m
3:  output: M
4:  begin
5:     while |M|>1 do
6:        T ← $\emptyset$
7:        for all ($M_i$, $M_j$) $\epsilon$ $M^2$ do
8:           compareAndMatch($M_i$, $M_j$)
9:           T ← T $\cup$ {($M_i$, $M_j$, simDeg($M_i$, $M_j$))}
10:       end for
11:       P ← maxWeightMatch(T)
12:       for all ($M_i$, $M_j$) $\epsilon$ P do
13:          m ← merge($M_i$, $M_j$, bestMatch($M_i$, $M_j$))
14:          M ← M\{$M_i$, $M_j$}
15:          M ← M $\cup$ {m}
16:       end for
17:    end while
18: end.
```

The steps of the algorithm are the following:

Step 1 (line 7-10): In this step, we first compare each pair of input models using the procedure *compareAndMactch()* (line 8) that implements our *compare* and *match* operators presented above. This procedure calculates the similarity degree for each pair of models and identifies the best matching between their elements. The similarity degree and the best matching are respectively returned by the two following functions: *simDeg()* and *bestMatch()*. Then, we arrange the similarity degree in *T* (line 9). Finally, we represent the similarity degrees between all pairs of models as a complete weighted graph in which vertices represent models, and each edge weight represents the similarity degree between a pair of models. For example, the graph depicted in Fig. 6 represents the comparison result of four models respectively denoted by M_1, M_2, M_3 and M_4.

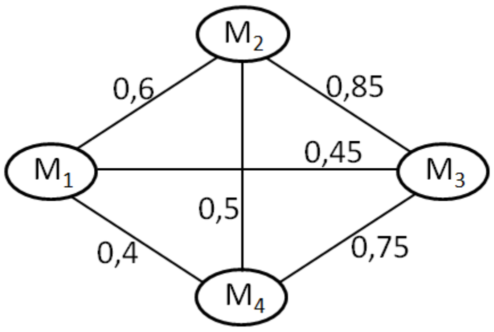

Fig. 6: Example of comparison result between a set of models

Step 2 (line 11): Our algorithm aims to combine as much as possible the most similar pairs of models. So, we try in this step to identify a set of *p* pairs of models, such that the sum of their similarity degrees is maximal, where *p = n/2* and *n* is the number of models in *M* (If *n* is impair, then *p* = (*n*-1)/2). We modeled the identification of this set as a maximum weighted matching problem in a general graph, where vertices represent models and weights represent similarity degrees between them. The maximum weighted matching problem is to find in an edge-weighted graph the matching of maximum weight. A matching in a graph *G* is defined as a subset of the edges of *G*, such that no two edges incident to the same vertex. The weight of a matching is the sum of the weights of its edges [34]. For example, all possible matchings of the graph depicted in Fig. 6 and their weights are presented in Table 5. The first matching has the maximum weight, and consequently the two pairs (M_1, M_2) and (M_3, M_4) are selected to be merged. In order to solve the maximum weighted matching problem, we use the Edmond's algorithm [28]. Based on *Gabow's* implementation of this algorithm [35], the *maxWeightMatch()* function identifies the set of *p* pairs of models and arranges the result in *P*.

Table 5: An example of possible matchings obtained when comparing a set of models

Matching	(M_1, M_2), (M_3, M_4)	(M_1, M_3), (M_2, M_4)	(M_1, M_4), (M_2, M_3)
Weight	1.35	0.95	1.25

Step 3 (line12-16): In this step, we combine each pair of models (M_i, M_j) from *P* using our *merge* operator. This latter employs the best match calculated by the *compareAndMactch()* procedure (in line 8), and returned by the *bestMatch()* function. Finally, we replace the two models M_i and M_j in *M* by the model resulting from their merging (line 14 and 15).

For example, the application of our algorithm on the models of Fig. 6 produces after the third step of the first iteration two models respectively obtained by merging M_1 with M_2, and M_3 with M_4. These two models will be merged in the second iteration to obtain a single model.

5.0 TOOL SUPPORT

We have implemented the approach discussed in this paper as a tool called 3M (Merging Many Models). Fig. 7 shows how this tool is applied to merge a set of models represented in XMI format. 3M consists of four modules: The first module allows us to (1) obtain the input models by parsing the XMI files, and (2) to convert them into our model representation (presented in Section 4.1). The second module allows us to compare each pair of models using our *compare* and *match* operators (described in Section 4.2). The third module allows selecting a set of pairs of models to be merged according to the algorithm described in Section 4.5. Each selected pair of models is merged by the fourth module using our *merge* operator (described in Section 4.4). The result is then used as input to the second module, and the process continues until obtaining a single model. The results of the second and third modules can be reviewed by the user.

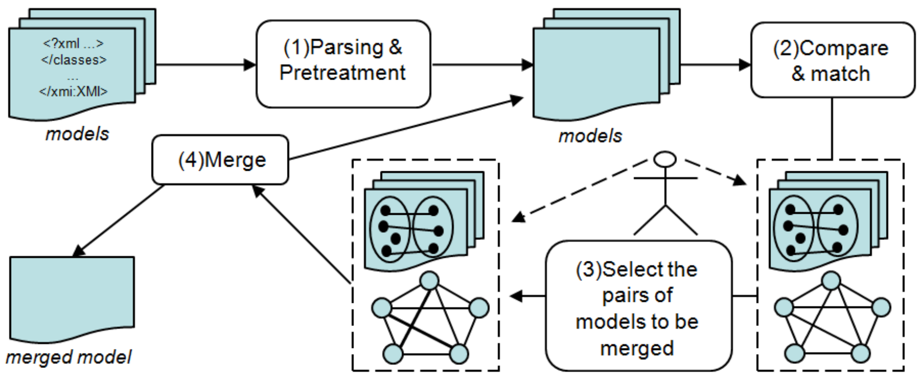

Fig. 7: Tool support overview

As we have previously seen, some configurations have to be done before using our model merging approach. In the current version of 3M, the user can use a configuration file in order to specify (1) the criteria weights of the element roles, (2) different thresholds, and (3) the parameter *N*[3] of the N-gram algorithm.

We have implemented our tool in Java based on a set of existing open source implementations, like for example the ws4j[4] (WordNet Similarity for Java) API used to implement the linguistic matching, and the JDOM2[5] (Java Document Object Model) API used to parse the XMI files. Our 3M tool consists of about 3400 lines of the code, and has a graphical interface allowing to visualize models in the form of graphs[6]. We have published 3M as a jar file with its documentation at: *https://drive.google.com/open?id=1sbr0KMrMV78v-9aDUUoyw07A2MnbOSjP*, and as an open-source project on GitHub at: *https://github.com/boubakir/ModelMerging/tree/master/ModelMerging/mmm*.

Our tool has been used to perform the empirical evaluation described in the next section.

6.0 EVALUATION AND DISCUSSION

This section is devoted to the evaluation of our approach and the discussion of its advantages and limitations. In order to assess the feasibility of our approach, we first calculated its complexity, and then empirically evaluated it using a set of UML class diagrams.

6.1 Complexity

We calculated the time complexity of our model merging algorithm as a function of the number of input models (denoted by n) and the size (the number of first-level elements, like for instance, classes in UML class diagram) of the largest model (denoted by k). In each iteration of our model merging algorithm, we first compare $n\times(n-1)/2$ pairs of models and find the best matching between the elements of each pair of them using the Hungarian method. Therefore, this latter which has time complexity $O(k^3)$ is called $n\times(n-1)/2$ times. Then, we identify the subset of pairs of models to be merged (pairs of models with the maximum sum of similarity degrees) using a single call of the Edmond's algorithm which has time complexity[7] $O(m^3)$, where m is the number of models in the current iteration. After each iteration, the number of models is divided by 2. Finally, after some calculations we obtain the time complexity of our approach which is polynomial both in n and k, and is bounded by $O(\log_2(n)\times(n^2\times k^3+n^3))$. This complexity is reasonable considering the fact that taking into account the quality of results when merging a set of model is a hard problem. This problem is reduced to the NP-hard problem of weighted set packing [36] in the case of simultaneous processing of all input models [9]. Moreover, the complexity of our approach is better than that of the approach presented in [9], which is considered as the closest approach to ours, and which has a complexity bounded by $O(n^4\times k^4)$.

[3] The N-gram algorithm calculates the similarity degree between two strings based on counting the number of their identical substrings of length *N*. In 3M, the default value of *N* is 3.
[4] https://code.google.com/archive/p/ws4j/
[5] http://www.jdom.org/
[6] The graphical interface is developed based on the GraphStream library which is available at: http://graphstream-project.org/
[7] This complexity concerns the implementation proposed by Gabow [35] and used in our work.

6.2 Evaluation

For simplicity, we divided the evaluation into two parts. The first part is devoted to our model merging algorithm, while the second part is devoted to evaluate our implementation of the three operators: *compare*, *match*, and *merge*. The evaluation is performed on an AMD A4-4300M APU 2.35GHz machine with 8GB of RAM.

6.2.1 Evaluation of the Model Merging Algorithm

In this section, we evaluate the ability of our model merging algorithm to produce better quality results. In order to do that, we applied it to four case studies and compared the results with three other pairwise model merging algorithms respectively denoted by *rand*, *ascOrd*, and *descOrd*. Each of these algorithms manipulates a pair of models in the same way as our algorithm, i.e., they use the same implementation of the three operators, *compare*, *match* and *merge* described in section 4. They only differ from our algorithm in the order of combining the set of input models. The first algorithm (*rand*) merges input models in a random order, i.e., it merges in the first step two randomly selected models, then merges the result with another randomly selected model, and so on, until merging all models. The second algorithm (*ascOrd*), merges the input models after arranging them in ascending order by their size (number of elements). Finally, the third algorithm (*discOrd*) performs the merge after arranging the input models in descending order. The case studies used are shown in Table 6. The first and the second one (*Hospital-v1* and *Hospital-v2*) are two different versions of *Hospital* system. While the third and the fourth one (*Warehouse-v1* and *Warehouse-v2*) are two different versions of Warehouse system. Each case study consists of a set of different variants of the same system. The first column of the table gives the number of variants (number of models) of each case study. The second column gives the number of elements (number of classes) in all models. The elements of the first version of each system are very varied in terms of their size relatively to those of the second version.

Table 6: The case studies of the first part of the evaluation

	Number of models	**Number of classes**
Hospital-v1	8	221
Hospital-v2	8	82
Warehouse-v1	16	388
Warehouse-v2	16	159

The evaluation is achieved as follows: First, we applied each algorithm on each case study, and then we measured the weight of the resulting models and the approximate execution time. Finally, we compared the results. An algorithm is considered to be good if it produces a model with a height value of weight. Similarly to [9], we performed the comparison by calculating for each case study, the percentage of the improvement (or degradation) that our algorithm provided compared by the other algorithms. The results are summarized in Table 7, where the first three columns present respectively the results obtained by the algorithms *rand*, *ascOrd*, and *desOrd*. The result obtained by our algorithm is presented in the fourth column. Finally, the last column presents the improvement (or degradation) percentage obtained by our algorithm over the best of the three other algorithms. The minus sign (-) precedes the percentage value in the case of degradation.

As illustrated in Table 7. For the first case (*Hospital-v1*), our algorithm outperforms both *rand* and *ascOrd* algorithms. It improves their results by respectively 1.43*%, and 10.10%*. However, it does not outperform *desOrd*, and presents a degradation of 3.77%. For the second (*Hospital-v2*), the third (*Warehouse-v1*), and the fourth (*Warehouse-v2*) case studies, our algorithm outperforms all other ones. More precisely, for the second case, it improves the results produced *by rand*, *ascOrd*, and *desOrd*, respectively, by 10.98%, 10.22%, and 1.34%. For the third case, it improves the results respectively, by 2.10%, 11.56%, and 0.67%. And finally, for the fourth case, it improves the results respectively, by 5.3 %, 16.28%, and 10.39%. For all case studies, our algorithm outperforms all other algorithms except the first case where it is surpassed by *desOrd.* Comparing our algorithm with this latter algorithm for all case studies, we found that it presents a degradation of 3.77% and three improvements: 1.34, 0.67, and 10.39.

Table 7: The results of the of the first part of the evaluation

		rand	**ascOrd**	**desOrd**	**Our algorithm**	**Impro/Degrad**
Hospital-v1	Weight	5.383	4.959	**5.666**	5.460	-3.77%
	Time	5s	5s	5s	21s	

Hospital-v2	Weight	1.293	1.302	1.416	**1.435**	1.34%
	Time	<1s	<1s	<1s	2s	
Warehouse-v1	Weight	1.333	1.220	1.352	**1.361**	0.67%
	Time	8s	8s	8s	49s	
Warehouse-v2	Weight	0.434	0.393	0.414	**0.457**	5.3%
	Time	1s	1s	1s	6s	

6.2.2 Evaluation of the Compare, Match and Merge Operators

This part of the evaluation aims principally to assess the ability of our approach to produce accurate results when manipulating a pair of models. More precisely, it assesses whether the elements of a pair of models are correctly matched. For this, we used two metrics: *precision* and *recall* that measure respectively quality and coverage. Precision is the ratio of correctly identified matches to the total number of identified matches. Recall is the ratio of the correctly identified matches to the total number of all correct matches. A matching technique is considered to be effective if it produces high precision and high recall. However, these two metrics are known to be inversely related [12]. High precision means that the number of incorrect matches is not high, and consequently, the user spends less effort to remove them. High recall means that the number of missing correct matches is not high, and consequently, the user spends less effort to find them.

The evaluation is performed by applying our implementation of the three operators to a set of model variants generated from three software product lines. These three families of systems are respectively called NotePad, GameOfLife and GPL. They are implemented in Java and are obtained from the FeatureIDE framework[8]. The evaluation is carried out as follows: First, we generated the large product variant from each family using the FeatureIDE framework. As result we obtained three Java applications. Then, we used reverse engineering technique to obtain three models from the Java source code. Some characteristics of these models are presented in Table 8. After that, for each of these models, we performed the following steps: (1) randomly generating two variants from the selected model, (2) applying our approach to merge these two variants, (3) calculating precision and recall. These steps are performed with different values of the threshold ranging from 0 to 1, and repeated fifty times for each of these values. Finally, we took the average of precision and recall values for each model and for each value of the threshold. The results are summarized in Fig. 8

To generate a variant of a model of Table 8, we automatically created a copy of it with two modifications. The first modification consists of randomly deleting up to 50% of the elements. The second modification consists of randomly renaming up to 50% of element names (class names, attribute names, and operation names). The renaming process consists of randomly changing until 50% of the characters of a string. During this evaluation, we used the criteria presented in Table 9. Furthermore, different roles take the same value of the threshold. For example, if the threshold is equal to 0.5, all the flowing roles have a threshold equal to 0.5: class, class attribute, class operation, parameter, class name, attribute name, attribute type, operation name, operation type, parameter name, and parameter type.

Table 8: The case studies of the second part of the evaluation

	Number of classes	**Number of Attributes**	**Number of operations**	**Number of all elements**
NotePad	9	79	55	498
GPL	14	37	143	859
GameOfLife	12	37	57	474

As shown in Fig. 8, for each of the three systems, our approach produces high precision. It is equal to 68% in the worse case. When the threshold increases, precision increases until reaching 100%, but the recall decrease considerably until reaching 0%. For thresholds below or equal to 0.5, recall is more than 60%.

In addition to the accuracy test, we assessed the ability of our approach to provide results in considerable time. For this, we applied it to pairs of models containing a different number of classes ranging from 10 to 100 classes. For each number of classes, we proceeded as follows: First, we randomly generated two models from the GPL system by

[8] https://featureide.github.io

duplicating the elements. Then, we executed our approach to this pair of models and calculated the execution time. We repeated the previous steps ten times, and finally, we took the average of the execution time. Table 10 shows the number of classes and the average number of all elements of the models used. The result is presented in Fig. 9. As we can see, the execution time increases considerably as the size of the models increases. This is mainly due to the use of the Hungarian method at different levels to improve the quality of matching.

Table 9: Example of criteria for UML classes

Element Role	Sub-element Role	Criterion	Weight
Class	Class name	Similarity of class names	0.5
	Class attribute	Similarity of attributes	0.25
	Class operation	Similarity of operations	0.25
Class at-tribute	Attribute name	Similarity of attribute names	0.5
	Attribute type	Similarity of attribute types	0.5
Class op-eration	Operation name	Similarity of operation names	0.5
	Operation type	Similarity of operation types	0.25
	Parameter	Similarity of parameters	0.25
Parameter	Parameter name	Similarity of parameter names	0.5
	Parameter type	Similarity of parameter types	0.5

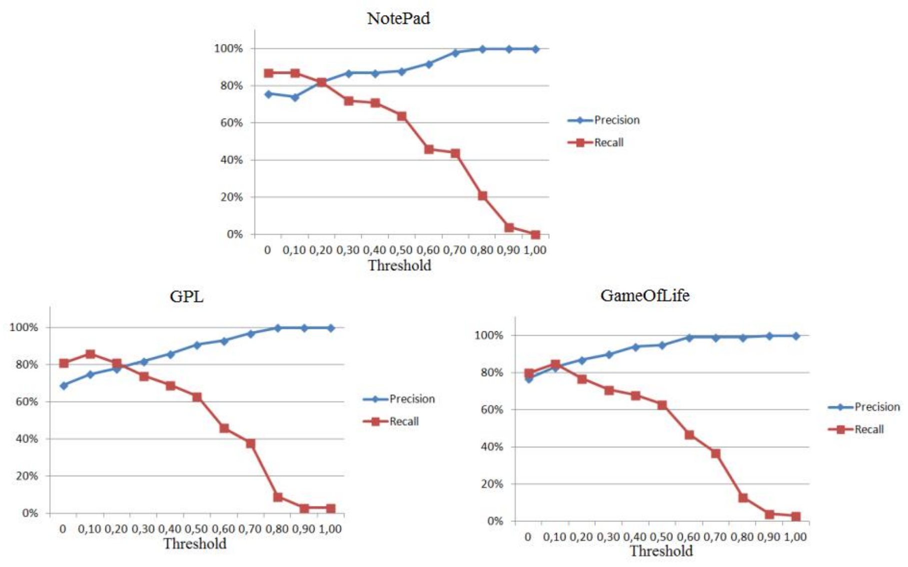

Fig. 8: Results of the precision and recall test

Table 10: Details about the models used in the execution time test

Number of classes	10	20	30	40	50	60	70	80	90	100
Average number of all elements	504	860	1512	1835	2460	2873	3245	3581	3970	4629

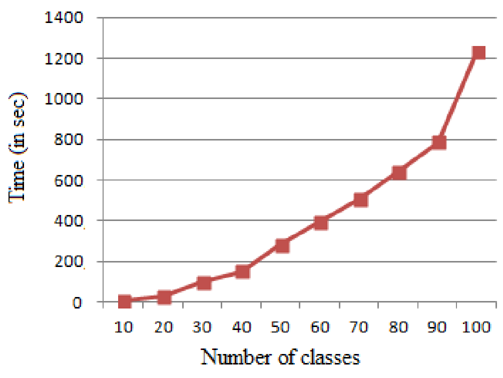

Fig. 9: Results of the execution time test

6.3 Discussion

We summarize in this section the main advantages and limitations of our approach.

6.3.1 Advantages

Our approach has the following advantages:

- It allows the user to merge a set of similar models in a fully automatic way. It requires only (1) the set of input models which are directly provided by the user or reverse engineered from source code, and (2) to set up some parameters.
- It uses a generic representation of models which allows it to be easily extendable to support a wide variety of models. Furthermore, it is customizable as it gives the user the possibility to set up a number of parameters, like for example the set of criteria and thresholds used in the model comparison step.
- The fact that it proposes to merge models in an incremental way provides the advantage of giving the user the possibility to intervene in order to review and, if necessary, adjust the result.
- It can be used to complement existing approaches. Users who already have experience with other pairwise model merging approaches can continue to use them to merge pairs of models and use our approach to enhance the quality of results by considering the order of merging.

6.3.2 Limitations

Our approach has some limitations:

- In order to calculate the similarity degrees of tuples and consequently to measure the result quality of a merge, we used a particular weight function inspired by [9] (the weight function presented in Section 3). Therefore, the result of our experimentation might not be generalized to other weight calculations.
- The current implementation of our approach has some limitations at the comparison level. Indeed, we did not consider the elements which are of reference types during the calculation of similarity degrees. This can influence the results especially in the case of models in which the elements of this type are very frequent, like for example, Statechart models.
- Our approach requires to apply several times the Hungarian method and the Edmond's algorithm that have polynomial complexity. This negatively influences the performance of the approach in term of execution time.

7.0 RELATED WORK

Several approaches have been proposed in the literature to address the problem of model merging. Most of these approaches lack generality or can be applied only to specific types of models. Some of them deal only with a sub-problem of model merging like, for example, model comparison. Furthermore, almost all of them focus only on merging two input models without proposing solutions for processing several (more than two) models.

The authors in [37] and [38] present two complete surveys on model comparison. Antoniol et al. address in [39] the problem of detecting differences between the releases of an object-oriented system. They propose to model the problem using a bipartite graph, where vertices represent classes of the two releases, and edges are weighted by the similarity degrees between classes. Kelter et al. address in [13] the problem of comparison between UML diagrams represented using XMI format. The authors in [14] focus also on UML diagrams. They propose UMLDiff which is an algorithm to detect differences between versions of UML class diagrams. Kpodjedo et al. propose in [15] an approach called MADMatch for solving the problem of detecting many-to-many matches in diagrams. They model the problem as a graph matching problem and solve it using tabu search. Nejati et al. present in [2] an approach for merging variant feature specifications represented using Statechart models. This approach is an extension of a previous approach proposed by the same authors in [12]. This latter allows merging hierarchical Statecharts models based on a heuristic operator for finding matches between models. A very simple approach for merging a pair of diagrams from the same model, and resolving conflicts is presented in [4]. The conflict problem is also considered by Dam et al. in [3]. They propose an approach for detecting and avoiding both syntactic and semantic consistencies that occur when merging versions of the same model. This approach is applied to model versions and their common ancestor using three-way merging.

The success of EMF has motivated the development of a number of approaches aiming at merging EMF models. For example, EMF Compare [40] is a meta-model independent approach for comparing versions of EMF models. It allows to compare two versions of a model which may be an instance of an arbitrary Ecore model. EMF Compare uses a variety of statistics and metrics in order to perform the comparison and represent the result as a model. EMF Diff/Merge [41] is a tool for merging EMF models based on user-defined policies. It ensures model consistency via predefined consistency rules. Westfechtel presents in [16] a formal approach for merging EMF models allowing both two- and three-way merging of two alternative versions. To ensure the consistency of the resulting model, the merge algorithm uses a set of rules allowing to detect and to resolve merge conflicts. The problem of merging EMF models is also addressed by the authors in [18]. In particular, they tackle a sub-problem of three-way model merging, which is merging of ordered collections. To perform model merging, they first propose to (1) represent each version of a collection as a linearly ordered graph. Then, (2) use a set of formula to combine the graphs that represent the collections. In order to create the merged collection, they propose to perform a generalized topological sort on the merged collection graph.

Contrary to our approach which aims to merge a set of models, the previous approaches are characterized by being limited to processing two models only. Some other approaches can be applied to a set of models. For example, the MoVaC approach [42] allows comparing a collection of model variants and identifying commonality and variability between them. This approach is designed to be generic and applicable to any EMF-based models. The main idea behind it is to divide each model variant into a set of atomic model elements according to MOF specification [43], then reuse the algorithm presented in [44] to identify commonality and variability. MoVaC allows to graphically represent the comparison result in the form of features, where each feature consists of a set of atomic model elements. Unlike our approach, MoVaC does not take into account the quality of merge. Mansoor et al. [45] formalize the model merging problem as a multi-objective problem based on NSGA-II algorithm [46], and propose an operation-based approach for model merging which considers the importance of operations in the merging process. This approach takes as input a set of model versions and their common ancestor, the list of applied operations and their importance scores, and produces as result a set of merging solutions. It aims to find the best trade of between minimizing the number of disabled operations and maximizing the number of important enabled ones. This approach is different from ours, as it is an operation-based approach, contrary to our approach which is a state-based approach. Moreover, as we have seen above, it takes as input not only a set of model variants as our approach does, but also their common ancestor, the list of applied operations and their importance scores.

Rubin and Chechik present in [9] an approach for merging together a set of (two or more) input models. The authors focus on the matching problem between the elements of the input models. They formulate this problem as a *weighted set packing problem*, and solve it by proposing a heuristic algorithm which simultaneously treats all input models. This work is similar to ours as it addresses the problem of merging not only a pair of models but many models, and takes into account the quality of merge. However, it performs the merging of the input models in completely different way from ours. In our work, input models are not simultaneously considered, but they are merged in a progressive and pairwise way, by tacking in to account the order of merging in order to enhance the quality of the result.

8.0 CONCLUSION AND FUTURE WORK

Model merging is an important and challenging task in Model-Driven Engineering. It has attracted a lot of attention in recent years. However, almost all existing approaches focus only on merging two model variants and do not consider the result quality when merging many (more than two) models.

This paper proposes a model merging approach that aims to obtain better quality results when merging a set of models in a pairwise way. Its main idea is to consider the order of merging the input models, and the way of matching the elements of each pair of models. The set of input models are merged using an iterative process, which is repeated until obtaining only one model. This process includes two basic steps: In the first step, we compare every pair of models in order to calculate the similarity degree between each two models and identify the best matching between their elements. We modeled the best matching identification problem as the maximum weighted matching in bipartite graphs and solved it in polynomial time using the Hungarian method. In the second step, we identify the most similar pairs of models and merge each of them. The identification of the most similar pairs of models is modeled as a maximum weighted matching problem in a general graph and solved in polynomial time using an implementation of the Edmond's algorithm. In order to make model comparison and matching more effective, we consider both typographic and linguistic matching.

Our approach is designed to be easily extendable to support a wide variety of models. Furthermore, our approach is customizable as it gives the user the possibility to set up several parameters like, for example, a set of criteria and thresholds.

We have implemented and evaluated our approach by applying it to a set of UML models, and the experiments show that it is promising. We have also published the tool implementing it as an open-source project on *GitHub* platform to allow others to use and extend it.

Our main direction for future work is to provide a generic solution for model merging. Therefore, we are interested in applying our approach to other types of models than UML models, and extending it to support any kind of EMF-based model. We plan also to provide a more complete implementation of our approach in the form of an Eclipse plugin.

REFERENCES

[1] M. Brambilla et a1., Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers, 2012

[2] S. Nejati et al., “Matching and Merging of Variant Feature Specifications”. IEEE Transactions on Software Engineering, vol. 38, 2012, pp. 1355–1375

[3] H. K. Dam et al., “Consistent Merging of Model Versions”. Journal of Systems and Software, vol. 112, 2016, pp. 137–155

[4] B. S. Thakare et al., “New Approach for Model Merging and Transformation”, in International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 2012

[5] H. Chong et al., “Composite-Based Conflict Resolution in Merging Versions of UML Models”, in 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Shanghai, China, 2016

[6] M. Sabetzadeh et al., “A Relationship-Driven Framework for Model Merging”, in MISE '07: Proceedings of the International Workshop on Modeling in Software Engineering, Minneapolis, MN, USA, 2007

[7] S. Melnik et al., Concepts And Algorithms. Springer, 2004

[8] K. Pohl et al., Software Product Line Engineering: Foundations, Principles and Techniques. Springer, New York, 2005

[9] J. Rubin et al., “N-Way Model Merging”, in ESEC/FSE, ACM, 2013, pp. 301–311

[10] J. Rubin et al., “Combining Related Products into Product Lines”, in FASE’12, Springer, 2012, pp. 285–300

[11] J. Rubin et al., “Quality of Merge-Refactorings for Product Lines”, in FASE'13 Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering, Springer, Heidelberg, 2013, pp. 83–98

[12] S. Nejati et al “Matching and Merging of Statecharts Specifications”, in ICSE 2007, IEEE, 2007, pp. 54–64

[13] U. Kelter et al., “A Generic Difference Algorithm for UML Models”, in Software Engineering, Vol. 64, 2005, pp. 105–116

[14] Z. Xing et al., “UMLDiff: An Algorithm for Object-Oriented Design Differencing”, in ASE, ACM, 2005, pp. 54–65

[15] S. Kpodjedo et al., “MADMatch: Many-to-Many Approximate Diagram Matching for Design Comparison”. IEEE Transactions on Software Engineering, Vol. 39, 2013, pp.1090–1111

[16] B. Westfechtel, “Merging of EMF Models”, in Software & Systems Modeling (SoSyM), Vol. 13, 2014, pp. 757–788

[17] B. Westfechtel, “A Formal Approach to Three-Way Merging of EMF Models”, in IWMCP '10 Proceedings of the 1st International Workshop on Model Comparison in Practice, Malaga, Spain, 2010, pp. 31–41

[18] F. Schwägerl et al., “A Graph-Based Algorithm for Three-Way Merging of Ordered Collections in EMF Models”. Science of Computer Programming journal, Vol. 113, 2015, pp. 51–81

[19] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/

[20] K. Altmanninger et al., “A Survey on Model Versioning Approaches”. International Journal of Web Information Systems, Vol. 5, 2009, pp. 271–304

[21] P. Brosch et al., “An Introduction to Model Versioning”, in proceedings of the 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM), 2012

[22] T. Mens, “A State-of-the-Art Survey on Software Merging”. IEEE Transactions on Software Engineering, Vol. 28, 2002, pp. 449–462

[23] X. Zhang et al., “Model Comparison to Synthesize a Model-Driven Software Product Line”, in SPLC, IEEE, 2011, pp. 90–99

[24] J. Martinez et al., “Bottom-Up Adoption of Software Product Lines - A Generic and Extensible Approach”, in SPLC, ACM, 2015, pp. 101–110

[25] A. Shatnawi et al., “Recovering Software Product Line Architecture of a Family of Object-Oriented Product Variants”. Journal of Systems and Software, Vol. 131, 2017, pp. 325–346

[26] M. Boubakir et al., “A Pairwise Approach for Model Merging”, in proceedings of MISC’16, Springer, 2016, pp. 327–340

[27] H. W. Kuhn, “The Hungarian Method for the Assignment Problem”. Naval Research Logistics Quarterly, Vol. 2, 1955, pp.83–97

[28] J. Edmonds “Maximum Matching and a Polyhedron with 0,1-Vertices”. Journal of Research National Bureau of Standards. Section B 69, 1965, pp. 125–130

[29] C. Ignat et al., “Operation-Based Versus State-Based Merging in Asynchronous Graphical Collaborative Editing”, in Workshop on Collaborative Editing, 2004

[30] OMG, http://www.omg.org/spec/XMI/2.1.1/

[31] A. Duley et al., “A Program Differencing Algorithm for Verilog HDL”, in proceedings of ASE’10, 2010, pp. 477–486

[32] C. Manning et al., Foundations of Statistical Natural Language Processing. MIT Press, 1999

[33] T. Pedersen et al., “WordNet:: Similarity - Measuring the Relatedness of Concepts”, in AAAI, 2004, pp. 1024–1025

[34] D. E. Drake et al., “A Linear-Time Approximation Algorithm for Weighted Matchings in Graphs”. ACM Transactions on Algorithms (TALG), Vol. 1, 2005, pp. 107–122

[35] H. N. Gabow, “An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs”. Journal of the ACM (JACM), Vol. 23, 1976, pp. 221–234

[36] E. M. Arkin et al., “On Local Search for Weighted k-Set Packing”. Mathematics of Operations Research, Vol. 23, no. 3, 1998, pp. 640–648

[37] M. Stephan et al., “A Survey of Model Comparison Approaches and Applications”, in 1st International Conference on Model-Driven Engineering and Software Development, INSTICC Press, 2013, pp. 265–277

[38] D. D. Kolovos et al., “Different Models for Model Matching: An Analysis of Approaches to Aupport Model Differencing” in proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM, IEEE, 2009, pp. 1–6

[39] G. Antoniol et al., “Maintaining Traceability Links during Object-Oriented Software Evolution”, in Software-Practice and Experience, Vol. 31, 2001, pp. 331–355

[40] C. Brun et al., “Model Differences in the Dclipse Modeling Framework”. UPGRADE, The European Journal for the Informatics Professional, Vol. 9, 2008, pp. 29–34

[41] EMF Diff/Merge: a Tool for Merging Models, https://wiki.eclipse.org/EMF_DiffMerge

[42] J. Martinez et al., “Identifying and Visualizing Commonality and Variability in Model Variants”, in proceedings of ECMFA, vol. 8569, ACM, 2014, pp. 117–131

[43] OMG: Meta Object Facility (MOF) Core Specification, http://www.omg.org/spec/MOF/2.0/

[44] T. Ziadi et al., “Feature Identification from the Source Code of Product Variants”. In: CSMR, 2012, pp. 417–422

[45] U. Mansoor et al., “MOMM: Multi-Objective Model Merging”. Journal of Systems and Software, Vol. 103, 2015, pp. 423–439

[46] K. Deb et al., “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II”. IEEE Transactions on Evolutionary Computation, Vol. 6, 2002, pp. 182–197