N-Way Model Merging

Project Report for CSC2125 Special Topics in Software Engineering: Modeling – Methods, Tools and Techniques

Julia Rubin
Computer Science Department,
University of Toronto, Canada
mjulia@il.ibm.com

ABSTRACT

We consider the problem of model merging. We focus on cases when several input models are merged into one, e.g., when combining a set of related products into a software product line representation. Most of the existing works on model merging focus on combining *two* input models, suggesting to merge n models by performing n-1 pairwise merges. In this paper, we show that combining n models *simultaneously* can produce a better result. We thus propose an *n-way* model merging framework and formally define its properties. We instantiate the framework using the Alloy Analyzer and provide two examples for demonstrating the differences between the pairwise and n-may merging. We conclude by discussing issues with the implementation of the framework the remaining steps.

1. INTRODUCTION

Model merging – combining information from several models into a single one – is widely recognized as an essential step in a variety of software development activities. These include reconciling partial (and potentially inconsistent) views of different stakeholders [12], uniting changes made in distinct branches of a software configuration management (SCM) system [6], and combining variants of related products into a single-copy software product line (SPL) representation [10].

Model merging has been extensively studied in the literature and frameworks to reason about different merging approaches have been proposed [2]. Yet, most of the existing works focus on merging *two* input models. It is often essential to merge more than two models together: for example, when reconciling views of multiple different stakeholders or combining related product variants into an SPL representation. In such cases, existing approaches suggest to merge n input models one by one, performing n-1 pairwise combinations [10]. Some approaches also suggest heuristics for the order in which to pick the pairs [11].

In this work, we show that merging n input models in a pairwise manner, even when using heuristics on the order in which models are merged, can lead to suboptimal results. Intuitively, this occurs because the pairwise operations can only find "locally" optimal solutions, loosing the "global picture". We thus suggest to merge n input models *simultaneously*, using *one n-way merge operation*, rather than applying n-1 pairwise merges one after another. We refine the model merging problem to consider n inputs and discuss issues related to its implementation.

Contributions of this work:

1. A formal definition of the n-way model merging framework.
2. A concrete instantiation of the framework, applicable for combining models of related product variants into a single-copy SPL representation.
3. An example-based comparison of the pairwise and n-way merging approaches.
4. A list of open questions and an outline of future research directions.

The rest of the paper is structured as follows. In Section 2, we give the necessary background on model merging. An example that showcases the process of merging three input models in a pairwise and an n-way manner is given in Section 3. In Section 4, we formally define the n-way merging framework and its concrete instantiation in Alloy for the SPL scenario. Section 5 focuses on analyzing the differences between the pairwise and the n-way approaches. Finally, Section 6 outlines future steps, while Section 7 summarizes the report.

2. BACKGROUND: MODEL MERGING

Following [10], model merging consists of three steps: *compare*, *match* and *merge*. These steps are described below.

Compare is a heuristic function that receives as input a pair of elements from the distinct input models M_1 and M_2 and calculates their similarity degree: a number between 0 and 1.

$$compare : M_1 \times M_2 \rightarrow [0..1]$$

Numerous specific implementations, analyzing structural and behavioral properties of the compared elements, exist. Most works calculate the similarity degree between two elements by comparing their corresponding sub-elements and weighting the results using empirically determined *weights* [13, 6, 8]. These weights represent the contribution of model sub-elements to the overall similarity of their owning elements. For example, a similarity degree between two classes can be calculated as a weighted sum of the similarity degrees of their names, attributes, operations, etc. Some works also utilize behavioral properties of the compared elements, e.g., dynamic behaviors of states in the compared state machines, reminiscent of deciding bisimilarity [8].

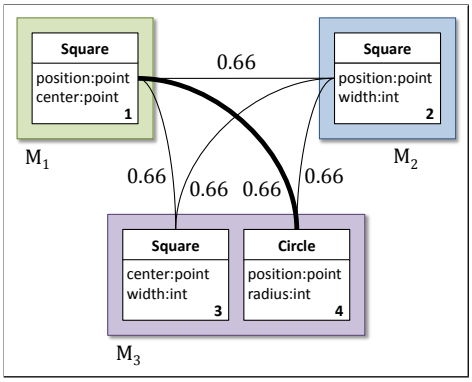


Figure 1: UML models M_1, M_2 and M_3. Numbers represent a result of pairwise comparisons.

Match is a heuristic function that receives pairs of elements from the distinct input models M_1 and M_2, together with their similarity degrees, and returns those pairs of model elements that are considered similar.

$$match : M_1 \times M_2 \times [0..1] \rightarrow M_1 \times M_2$$

Most implementations of *match* use empirically assigned *similarity thresholds* to decide such similarity. More advanced approaches, e.g., [3], rely on bipartite graph matching [7] to determine corresponding elements.

Finally, *merge* is a function that receives two input models M_1 and M_2, together with pairs of their matched elements and returns a merged model M that combines elements of the input in a prescribed way.

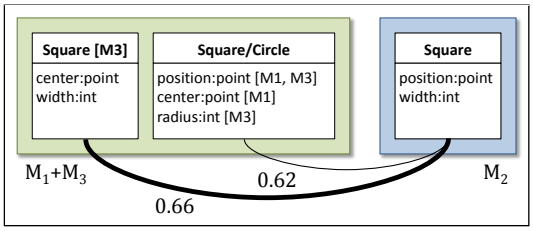
$$match : M_1, M_2, M_1 \times M_2 \rightarrow M$$

Specific merge algorithms define how to treat the matched and unmatched elements [2]. For example, the *union merge* approach [12] assumes that matched elements are *complementary* and should be unified in the produced result, while unmatched elements are copied to the result as is. It is also possible to produce a result that includes only those parts of the input models on which the matched elements *agree*. Yet another approach is to combine the input model elements as in union-merge, while keeping track of and explicating the origin of each element [10] (the *annotative* SPL merge approach).

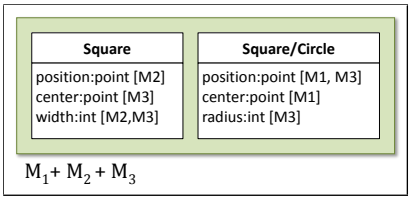
3. MOTIVATING EXAMPLE

We use the toy example of three UML models in Figure 1, inspired by [9], to illustrate the n-way approach to model merging. The first model, M_1, contains a single UML class `Square` (element #1) which has two attributes: `position`, specifying the location of the square's top left corner on the screen and `center`, specifying the location of its center. These two attributes uniquely describe the dimensions and the placement of the square. The second model, M_2, also contains a `Square` class (element #2). However, in this class, the dimensions and the placement of the square are described by the `position` and the `width` attributes. Finally, M_3 contains two elements: the `Square` class (element #3) described by the `center` and the `width` attributes, and the `Circle` class (element #4) described by the `position` of its highest point and its `radius`.

Comparing each pair of elements from the distinct models produces the similarity degrees shown as decimal numbers on the arcs connecting the corresponding elements. The specific details of the compare heuristic that we used to produce these numbers are not import at this point; these calculations are discussed in details in



(a) Merging M_1 and M_3.



(b) Merging M_1+M_3 and M_2.

Figure 2: A pairwise merge of the models M_1, M_2 and M_3 in Figure 1.

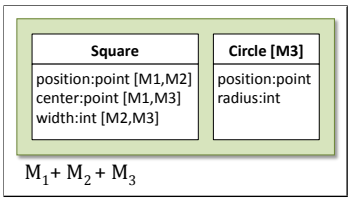


Figure 3: Another possible merge of the models M_1, M_2 and M_3 in Figure 1.

Section 4. However, it is important to notice that if we choose to exclude class names from comparison, all elements are pairwise similar. That is, all pairs conform to the same "pattern": two elements of the pair share one common attribute and contain one proprietary attribute each.

When considering only pairwise similarities between elements, one can choose to start from matching and merging element #1 from M_1 with element #4 from M_3 (as indicated by the bold arc in Figure 1), producing the merged model M_1+M_3 shown in the left part of Figure 2(a). The elements of this model are annotated by their origins. These elements are then compared to the `Square` class of M_2 (element #2) using the same compare heuristic, and the `Square` class of M_2 is matched and merged with the `Square` class of M_1+M_3 (as indicated by the bold arc in Figure 2(a)). The result of this merge is shown in Figure 2(b). Again, the elements of the resulting model M_1+M_2+M_3 are annotated by their original models.

Inspecting the three input models in Figure 1 more closely reveals that only the `Circle` class of M_3 (element #4) has the `radius` attribute. Moreover, the other three classes overlap on two out of three attributes with the others. Thus, a merge that combines these three `Square` classes together, as shown in Figure 3, might be better than the one in Figure 2(b). Indeed, it involves less annotations, contains less attributes and overall looks simpler. We also quantitatively confirm this intuition later in Section 5.

This means that instead of combining element #1 of M_1 with element #4 of M_3, as decided earlier, one should have chosen to combine it with element #3. Making this decision was impossible without having the "global picture", when considering elements in a pairwise manner only.

As discussed in Section 1, some approaches for model merging propose heuristics for picking the order in which input models are

merged [11]. It is plausible to assume that one of the heuristics could work for the example in Figure 1: the problem of merging elements #1 and #4 could be avoided if models M_1 and M_2 would be merged first and then compared to M_3. However, in Section 5, we show that applying such heuristics is not possible in a general case.

4. N-WAY MODEL MERGING

In this section, we refine the definition of merging given in Section 2 to deal with n input models. First, we formally specify the n-way counterparts of the *compare*, *match* and *merge* operators (Section 4.1). Then, we provide a concrete realization of these operators, designed with the goal of merging related product variants into SPL representations (Section 4.2).

4.1 The Framework

The main difference between the pairwise and the n-way merging approaches is the number of elements that are simultaneously compared, matched and merged together. While the first case operates on pairs of elements from distinct models, the second considers *n-tuples* of elements.

DEFINITION 1. *For a set of models $M = M_1, M_2, \dots M_n$, an n-tuple (*or a *tuple,* for simplicity*) is a set $t \in (M_1 \cup \{\emptyset\}) \times (M_2 \cup \{\emptyset\}) \times \dots \times (M_n \cup \{\emptyset\})$ of two or more elements from distinct models. That is, a tuple t satisfies the following two properties:*

(a) $|t| > 2$.

(b) $\forall e_i, e_j \in t; M_i, M_j \in M \mid e_i \in M_i \wedge e_j \in M_j \wedge e_i \neq e_j \Rightarrow M_i \neq M_j$.

For the example in Figure 1, a tuple can consist of element #1 from M_1, element #2 from M_2 and element #4 from M_3. We denote such tuple as (1,2,4). Another tuple could be (1,2). Yet another one is (1,2,3). (1,3,4) is not a valid tuple though, as it contains two elements from M_3.

In what follows, let T denote the set of all valid tuples for models $M = M_1, M_2, \dots M_n$. The size of T can be calculated using the formula below, which "chooses" an element from each model, including choosing none, but disallows tuples of size one or zero.

$$|T| = (|M_1| + 1) \times (|M_2| + 1) \times \dots \times (|M_n| + 1) - (|M_1| + |M_2| + \dots + |M_n| + 1)$$

The goal of *compare* is then to assign a similarity measure to a given tuple $t \in T$.

DEFINITION 2. Compare *is a function that, given a tuple $t \in T$, returns the similarity measure for its elements – a number in the range of $[0..1]$. The larger the number is, the more similar to each other the elements of t are considered to be.*

Match considers the compared tuples, and selects those that are deemed similar. As in the pairwise case, matching can assume an empirically set threshold $\mathcal{S}$ – elements that are below the threshold are considered highly dissimilar and should never be matched.

The set of tuples produced by *match* should be disjoint – it is not possible to match an element in multiple ways, i.e., to more than one element of any model. For the example in Figure 1, *match* can output the tuple (1,2,3) or (1,2,4), but not both, since otherwise element #1 would be matched to both element #3 and #4 of M_3.

DEFINITION 3. *Let $s(t)$ denote the similarity degree of a tuple $t \in T$, as calculated by compare, and let $\mathcal{S} \in [0..1]$ be the similarity threshold. Then,* match *is a function that returns a set of matches $\hat{T} \subseteq T$ that satisfy the following properties:*

```
1  enum Property { A, B, C, D, E, F, G, H, I, J, K, L, M,
2                  N, O, P, Q, R, S, T, U, V, W, X, Y, Z }
3  sig Element {properties : set Property}
4  abstract sig Model {
5     elements: set Element
6  }
7  {
8     #elements > 0
9  }
10
11 abstract sig InputModel, OutputModel extends Model {}
```

Figure 4: Model and model elements in Alloy.

```
1  one sig M1, M2, M3 extends InputModel {}
2  one sig M extends OutputModel {}
3
4  one sig m1Square, m2Square,  m3Square, m3Circle extends Element{}
5
6  fact elementProperties {
7     m1Square.properties = {A} + {B}
8     m2Square.properties = {A} + {C}
9     m3Square.properties = {B} + {C}
10    m3Circle.properties = {A} + {D}
11 }
12
13 fact elements {
14    M1.elements={m1Square}
15    M2.elements={ m2Square}
16    M3.elements={m3Square} + {m3Circle}
17 }
```

Figure 5: The example in Figure 1 in Alloy.

(a) $\forall \hat{t} \in \hat{T} \mid s(\hat{t}) \geq \mathcal{S}$.

(b) $\forall \hat{t}_i, \hat{t}_j \in \hat{T}; e_i \in \hat{t}_i; e_j \in \hat{t}_j \mid \hat{t}_i \neq \hat{t}_j \Rightarrow e_i \neq e_j$.

Elements of the matched tuples are combined with each other using the *merge* function. The n-way merging framework does not prescribe any particular way of combining the matched elements – any of the approaches discussed in Section 2 can be applied. Figure 2(b) shows a possible result of merging the input models in Figure 1, when *merge* assumes annotative SPL union-merge semantics and *match* produces two tuples: (1,4) and (2,3). Figure 3 shows another possible result, obtained for the case when *match* produces the tuple (1,2,3).

Discussing and comparing different possible realizations of the model merging framework, i.e., different implementations of the *compare*, *match* and *merge* operators, is orthogonal to our work and thus is out of the scope of this paper. Moreover, we believe that there cannot exist one preferable way to realize the operators. Instead, their realization is domain-specific and is driven by the *goal* of the merging process.

Yet, in order to carry out the discussion on the pairwise and n-way merging approaches and to compare the two, we need to pick an operational definition of the above operators. In the next section section, we present an implementation that we have chosen for this work.

4.2 A Realization of the Framework

Our implementation of *compare*, *merge* and *match* is aligned with our broader research agenda that includes using model merging for combining related product variants into annotative SPL representations [5, 1, 10]. In such representations, a set of related product variants is captured by a single model in which elements are "tagged" by features – an approach similar to preprocessor directives. In the simplest form, merging produces a feature for each input product and uses it to annotate all elements which originated from that product [10], like we did in Figures 2(b) and 3.

We instantiate the chosen implementation using the Alloy Analyzer [4] – a tool for declarative specification and SAT-based anal-

```
1  sig Tuple {
2      elements: set Element,
3      weight : Int,
4      union: set Property
5  }
6  {
7      #elements > 1
8      all e: elements | some m: InputModel | e in m.elements
9      all disj e1, e2: elements |
10         {all m1, m2: Model | {e1 in m1.elements and e2 in m2.elements => m1!=m2}}
11     all p: Property | {p in union <=> (some e: elements | {p in e.properties})}
12     weight = (sum e: elements | #e.properties).mul[10].div[#union]
13 }
```

Figure 6: Compare in Alloy.

```
1  sig Solution {
2      tuples: set Tuple,
3      weight: Int
4  }
5  {
6      #tuples > 0
7      all disj t1, t2: tuples | { #(t1.elements & t2.elements)=0} //disjoint
8      all t: Tuple |
9          { not t in tuples =>some t1: tuples |
10             {#(t.elements & t1.elements) > 0}} //maximal
11     weight = (sum t: tuples | t.weight).div[#tuples]
12 }
13 pred match[s: Solution] {
14     no o: Solution |   {o.weight > s.weight or
15                         (o.weight = s.weight and #o.tuples < #s.tuples)}
16 }
```

Figure 7: Match in Alloy.

```
1  pred merge[m: OutputModel] {
2      one s: Solution | match[s] and
3      all e: Element | ( (one im: InputModel | e in im.elements ) =>
4          (one t: s.tuples | e in t.elements) or
5          ((no t: s.tuples |e in t.elements) and e in m.elements)) and
6      all t:s.tuples | one e: m.elements | t.union = e.properties and
7      all e: m.elements |  (one t: s.tuples | t.union = e.properties) or
8          ((no t: s.tuples | e in t.elements) and one im: InputModel | e in im.elements)
9  }
```

Figure 8: Merge in Alloy.

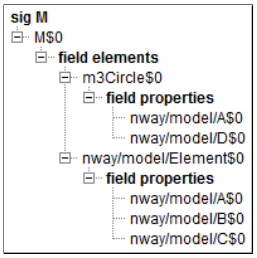


Figure 9: The merge produced by Alloy for models in Figure 1.

ysis of models. Figure 4 presents our simplified representation of models and model elements. In this representation, each element can have up to 26 different properties, represented as letters A to Z (lines 1–2). These properties encode, for example, UML class attributes, methods, etc., and are used for comparing elements to each other. A model is a non-empty set of elements (lines 4–9). We explicitly distinguish between input and output models (line 11).

Figure 5 uses this representation to encode the three input models in Figure 1 and one output model (lines 1–2). It declares four elements, corresponding to the elements #1–4 in Figure 1 (line 4), and uses the properties A, B, C and D to represent their attributes (lines 6–11). For example, the attribute `position` of elements #1, #2 and #4 in Figure 1 is represented by the property A, while the attribute `center` of element #4 is represented by the property D. Finally, we assign elements to their corresponding models (lines 13–17). We use this data representation to define *compare*, *match* and *merge* below.

Compare. Figure 6 specifies a tuple in Alloy. Similar to Definition 1, it states that a tuple is a set of more than one element (lines 2 and 7). It also states that the tuple's elements originate from distinct input models (lines 8–10).

Each tuple stores its *weight* (line 3) – the result of applying *compare* for the tuple, as per Definition 2. We define compare as *the total number of properties of all tuple elements divided by the number of properties in their union*. We normalize the result by the number of input models:

$$\text{compare}(t) = \frac{\sum_{e \in \text{t.elements}} |\text{e.properties}|}{|M| \times |\bigcup_{e \in \text{t.elements}} \text{e.properties}|}$$

The union of properties of the tuple elements represents the union-merge for the tuple (lines 4 and 11). Thus, our compare function essentially calculates the "distance" of the tuple elements from their potential union-merge. We choose this heuristic as it allows to minimize the number of annotations in the produced annotative product line representation [10].

For example, the result of applying compare on the tuple (1,2) that contains the `Square` classes from models M_1 and M_2 is $\frac{2+2}{3*3} = 0.44$: each element has two attribute and there are three attributes in their union – `position`, `center` and `width`. We normalize by the number of input models, i.e., three. The result of applying compare on the tuple (1,2,3) is $\frac{2+2+2}{3*3} = 0.66$, while the tuple (1,2,4) results in $\frac{2+2+2}{4*3} = 0.5$.

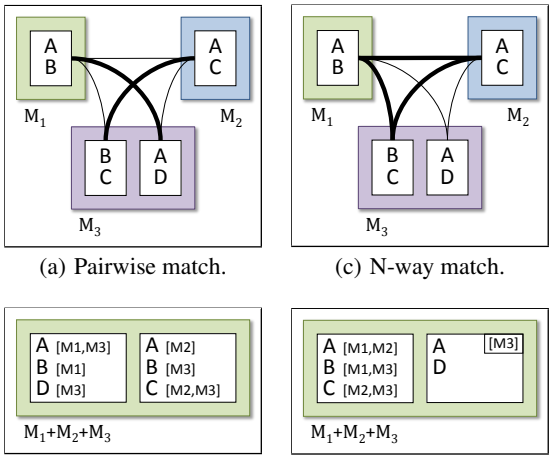
Note that even though the first tuple contains only two elements, we still normalize by the total number of input models, i.e., three. This allows "fair" comparison between tuples containing different number of elements: in some cases, merging only two elements without adding a third one is more beneficial (if the third element is radically different from the first two). In other cases, adding more elements improves the weight of a tuple (if they are highly similar to the existing tuple's elements).

Line 12 in Figure 6 calculates the tuple's weight. Since Alloy does not support floating point numbers, we limit the precision to one decimal digit and do not normalize by the number of models. This does not contradict the above discussion about the "fairness" of comparison but merely means that for three input models, our implementation produces results in the range of [0..30] rather than [0..1].

Match. The code in Figure 7 searches the space of different possible matches and then selects the best one. A possible match, referred to as a `Solution`, has a non-empty set of tuples (lines 2 and 6) and a weight (line 3). As in Definition 3, all tuples are disjoint (line 7). Also, it is not possible to add additional tuples to the solution without violating the above disjoint constraint (lines 8–11). This means that we attempt to find matches for as many elements of the input models as possible.

If matching elements with low similarity degree is undesirable, a similarity could be used to filter out tuples that correspond to such elements, e.g., all tuples with a weight below 15. Even though our implementation does not use any similarity threshold at this point, adding it to the statement in lines 8–11 is trivial.

The weight of a possible solution (line 3) is calculated by averaging weights of all its tuples (line 11). The `match` predicate (lines 13–16) then chooses a solution with the maximal possible weight. If there is more than one such solution, it prefers that with the minimal number of tuples (line 15): everything else been equal, we aim at more "tight" matches, minimizing the number of elements in the resulting merged model. Even with these restrictions, multiple matches are possible and we pick any.



(a) Pairwise match. (c) N-way match.

(b) Pairwise merge. (d) N-way merge.

Figure 10: Possible matches / merges for the models in Figure 1.

Merge. Figure 8 defines the *merge* operator. It picks the match produced by the *match* operator (line 2) and ensures that each input model element is either part of the matched tuple or is copied to the result as is (lines 3–5). It also ensures that the output model has one element for each of the matched tuples (line 6). The properties of that element are a union of properties of all elements in the tuple, as defined in line 4 of Figure 6. Moreover, each element of the resulting model either comes from a tuple in the match or is an unmatched element that is not included in any tuple (lines 7–8). This implementation corresponds to the definition of the union-merge approach [12]. Extending it to also "tag" elements of the resulting model by their origins, like in Figure 3, is trivial.

Figure 9 shows the result of merging the models in Figure 1, as produced by Alloy. Exactly as the result in Figure 3, it has the `Circle` element that was transferred to the result as is and an element that is obtained by merging the `Square` classes together. The first one has two properties, represented by A and D, while the second has three – A, B and C.

5. COMPARING THE PAIRWISE AND N-WAY MERGING ON EXAMPLES

In this section, we use two examples to illustrate the pairwise and the n-way model merging and compare the approaches to each other. We first analyze the example in Figure 1 and then further generalize it to gain better understanding of the differences between the approaches. For simplicity of the discussion, we represent model elements using their abstract properties (A to Z), as in the Alloy implementation (see Section 4.2). We also name the elements using these properties. This abstract representation allows us to focus on those details that are essential for the discussion.

Example 1. Figure 10(a) shows the representation of the three input models in Figure 1. As in that figure, the similarity degree between each pair of elements is $\frac{2+2}{3*2} = 0.66$ (we normalize here by 2 as we only consider two input models). Also, like in the pairwise merging described in Section 3, element AB is matched with AD, while AC is matched with BC. These matches are represented with bold lines connecting the corresponding elements. Using the n-way merging terminology from Section 4.2, this matching corresponds to a solution with two tuples – (AB, AD) and (AC, BC). The weight of each tuple is $\frac{2+2}{3*3} = 0.44$. Thus, the weight of the entire solution (the average weight of the tuples) is 0.44 as well. The corresponding merge is shown in Figure 10(b).

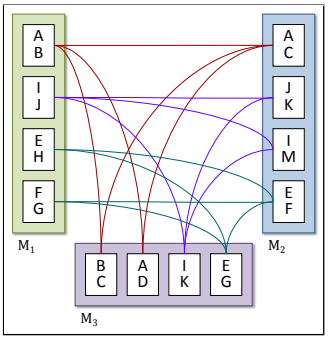
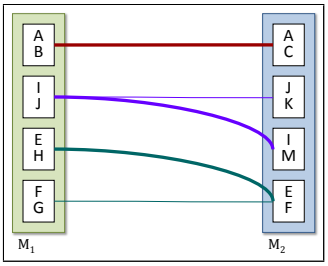
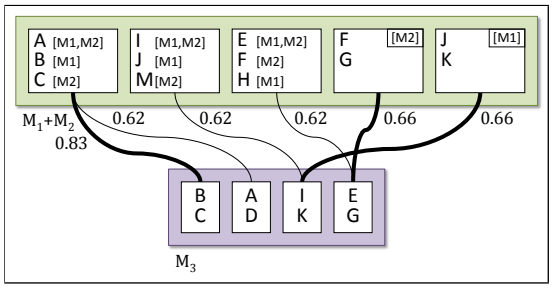


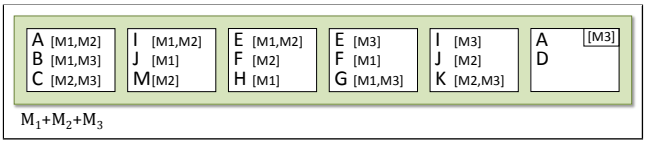
Figure 11: Example 2 – pairwise similar input models.



(a) Matching for M_1 and M_2.



(b) Merging M_1 and M_2.



(c) Merging M_1+M_2 and M_3.

Figure 12: A pairwise merge of the models M_1, M_2 and M_3 in Figure 11.

As discussed in Section 3, this solution is not the optimal one. Indeed, the solution shown in Figure 10(c) contains a single tuple, (AB, AC, BC), whose weight is $\frac{2+2+2}{3*3} = 0.66$. Thus, the weight of the entire solution is 0.66, which is larger than 0.44. This solution is found by our n-way merge implementation. The corresponding merge is shown in Figure 10(d).

A "smart" heuristic for the ordering of pairwise merges could suggest to match and merge the models M_1 and M_2 first, before considering M_3. In that case, pairwise merging would produce the result equivalent to the n-way algorithm. However, in some cases, input models are pairwise similar, thus heuristics on the order of merging cannot be helpful. The example below demonstrates one such case.

Example 2. Figure 11 generalizes the example in Figure 10(a). We now have four elements in each model. There are two types

of elements pairs – those that share one common property (e.g., AB and AC) and those that share none (e.g., AB and JK). Pairwise similarity measure between the pairs of the first type is $\frac{2+2}{3*2} = 0.66$, while pairs of the second type are $\frac{2+2}{4*3} = 0.33$ similar. In Figure 11, to avoid cluttering, we only show arcs for the pairs of the first type; pairs of the second type are disregarded by matching in any case, as their similarity measure is lower.

Each pair of models shares the same "similarity pattern", shown in Figure 12(a) for models M_1 and M_2: there are five possible pairs of elements considered by the *match*. The first pair is "exclusive" – its elements (e.g., AB and AC) are not part of any other possible match. They will be matched to each other. The next two pairs share an element in the first model (e.g., IJ). This element can thus be matched to either one of the two elements in the second model (e.g., either JK or IM). However, pairwise, it is impossible to see which of these two elements should be preferred. The last two pairs have the "inverse" issue: an element from the second model (e.g., EF) can be matched to one of the two elements in the first (e.g., either EH or FG). Again, it is impossible to see which of these two elements should be preferred when considering pairwise comparisons only.

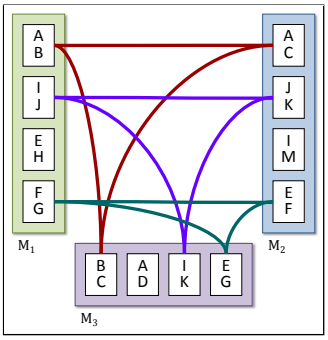
Since each pair of models exhibit the same similarity "pattern", the decision in which order to pick models for pairwise merging becomes arbitrary. Further, picking a match candidate out of two possible options is also an arbitrary decision. Thus, one could decide to start from the models M_1 and M_2, and produce the matches marked with the bold lines in Figure 12(a): AB and AC, IJ and IM, EH and EF. The corresponding merge is shown in the upper part of Figure 12(b).

The elements of the produced model M_1+M_2 are further matched with the elements of M_3, and their similarity measures are shown on the corresponding arcs in Figure 12(b). Again, we only show the results of comparison for pairs that share at least one common property. Matched pairs are marked with bold lines: ABC is matched with BC, FG with EG and JK with IJ. The produced merge is shown in Figure 12(c).

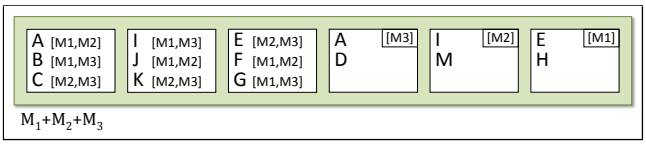
It can be seen that the n-way solution that corresponds to the merge in this figure has five tuples: (AB, AC, BC), (IJ, IM), (JK, IK), (EH, EF) and (FG, EG). The weight of the first tuple is $\frac{2+2+2}{3*3} = \frac{2}{3}$, while the weight of the other four is $\frac{2+2}{3*3} = \frac{4}{9}$. Thus, the overall weight of the solution is $\frac{\frac{2}{3}+4*\frac{4}{9}}{5} = \frac{22}{45} = 0.48$. However, when looking at all three models simultaneously, it becomes apparent that the element IJ should better be matched with JK rather than IM, because then they later can be matched with IK, constituting a tuple (IJ, JK, IK). The property M is an "outlier" and matching with the only element that contain this property will decrease the quality of the final result. The same reasoning holds for the element EF – it should be matched with FG rather than with EF.

Figure 13(a) shows such matching, produced by our n-way match algorithm. It has three tuples (AB, AC, BC), (IJ, JK, IK) and (FG, EF, EG). Each tuple's weight is $\frac{2+2+2}{3*3} = \frac{2}{3}$, and thus the weight of this solution is $\frac{2}{3}$ as well, which is larger than $\frac{22}{45}$ of the previous one. Figure 13(b) shows the corresponding merge. Apart of the quantitative comparison between the two, it can be seen that this result is visually simpler.

Conclusion. We showed that n-way merging produces results that are better than those obtained by the pairwise approach. We also showed that a heuristic for picking the order in which models are merged can fail if models are pairwise similar. Even though we do not theoretically compare the pairwise and the n-way approaches for model merging, our example shows that the pairwise approach is *at least* by $\frac{11}{15}$ worse than the n-way counterpart ($\frac{22}{45}$ vs $\frac{2}{3}$).



(a) N-way matching.



(b) N-way merging.

Figure 13: An n-way merge of the models M_1, M_2 and M_3 in Figure 11.

6. OPEN ISSUES AND FUTURE WORK

The main problem of our Alloy implementation is its scalability: the implementation will clearly not work for input models of a reasonable size. In fact, any implementation that is based on enumerating the tuples and constructing all possible solutions will not scale well, as these are exponential to the number of input model elements. We thus need to devise efficient exact or approximate solutions for the n-way merging. Moreover, for approximate solutions, it is important to show that they perform better than the pairwise merging approach – a reasonable approximation as well.

Specifically, the following three steps are main tasks of our future work:

1. Theoretically evaluate the approximation factor of the pairwise merging solution.
2. Devise an efficient n-way merging algorithm and theoretically evaluate it against the pairwise approach.
3. Implement the algorithm from step 2 and empirically evaluate it against the pairwise solution on a large set of real and randomly generated case studies.

7. SUMMARY AND CONCLUSIONS

In this report, we discussed the problem of merging n input models. We showed that the pairwise approach to merging the models one-by-one can miss the "global picture" and produce a result that is inferior to an approach that considers all input models *simultaneously*. We thus defined an n-way modeling framework that consist of n-way compare, match and merge operators. We formally specified the properties of these operators and instantiated them for the case of combining related products into an SPL representation, using the Alloy Analyzer. We used the operational implementations of the framework to analyze the differences between the pairwise and the n-may merging approaches on two concrete examples. We also showed that heuristics for choosing the order in which input models are merged can be ineffective, as in some cases input models are pairwise similar. Finally, we discussed issues related to the implementation of the n-way merge and outlined future steps for continuing this work.

8. REFERENCES

[1] Q. Boucher, A. Classen, P. Heymans, A. Bourdoux, and L. Demonceau. Tag and Prune: a Pragmatic Approach to Software Product Line Implementation. In *Proc. of ASE'10*, 2010.

[2] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *Proc. of GaMMa'06*, pages 5–12, 2006.

[3] A. Duley, C. Spandikow, and M. Kim. A Program Differencing Algorithm for Verilog HDL. In *Proc. of ASE'10*, pages 477–486, 2010.

[4] D. Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[5] C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proc. of GPCE Wrksp. on Modul., Comp. and Gen. Tech. for PLE (McGPLE)*, pages 35–40, 2008.

[6] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, volume 64 of *LNI*, pages 105–116. GI, 2005.

[7] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[8] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proc. of ICSE'07*, pages 54–64, 2007.

[9] D. Ohst, M. Welle, and U. Kelter. Merging UML Documents. Technical report, Department of Electrical Engineering and Computer Science, University of Siegen, Germany, 2004. Internal Report.

[10] J. Rubin and M. Chechik. Combining Related Products into Product Lines. In *Proc. of FASE'12*, pages 285–300, 2012.

[11] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *Proc. of FASE'13*, 2013. To appear.

[12] M. Sabetzadeh and S. Easterbrook. View Merging in the Presence of Incompleteness and Inconsistency. *Requirement Engineering*, 11:174–193, June 2006.

[13] Z. Xing and E. Stroulia. UMLDiff: an Algorithm for Object-Oriented Design Differencing. In *Proc. of ASE'05*, pages 54–65, 2005.