N-Way Model Merging

Julia Rubin
IBM Research at Haifa, Israel
and
University of Toronto, Canada
mjulia@il.ibm.com

Marsha Chechik
University of Toronto, Canada
chechik@cs.toronto.edu

ABSTRACT

Model merging is widely recognized as an essential step in a variety of software development activities. During the process of combining a set of related products into a product line or consolidating model views of multiple stakeholders, we need to merge multiple input models into one; yet, most of the existing approaches are applicable to merging only two models.

In this paper, we define the n-way merge problem. We show that it can be reduced to the known and widely studied NP-hard problem of weighted set packing. Yet, the approximation solutions for that problem do not scale for real-sized software models. We thus evaluate alternative approaches of merging models that incrementally process input models in small subsets and propose our own algorithm that considerably improves precision over such approaches without sacrificing performance.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Management

Keywords

Model merging, combining multiple models, weighted set packing.

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 [20], composing changes made in distinct branches of a software configuration management (SCM) system [13], and combining variants of related products into a single-copy software product line (SPL) representation [10, 7, 18].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491446

Merging is usually seen as a combination of three distinct operators – *compare*, *match* and *compose* – all of which have been extensively studied in the literature [4, 18]. Yet, while many aforementioned activities require combining *multiple* sources of input together, the existing work mostly focuses on merging just *two* input models [21, 13, 16]. A straightforward approach to merging multiple inputs is then to do so in the pairwise manner, i.e., merge n models using $n-1$ pairwise operations. *A subset-based approach* further generalizes on that, allowing to fix a certain number of input models that are processed in each iteration. However, the quality of the result produced using subset-based approaches, as well as the effect of the chosen subset size or the order in which the input models are picked, remains unclear.

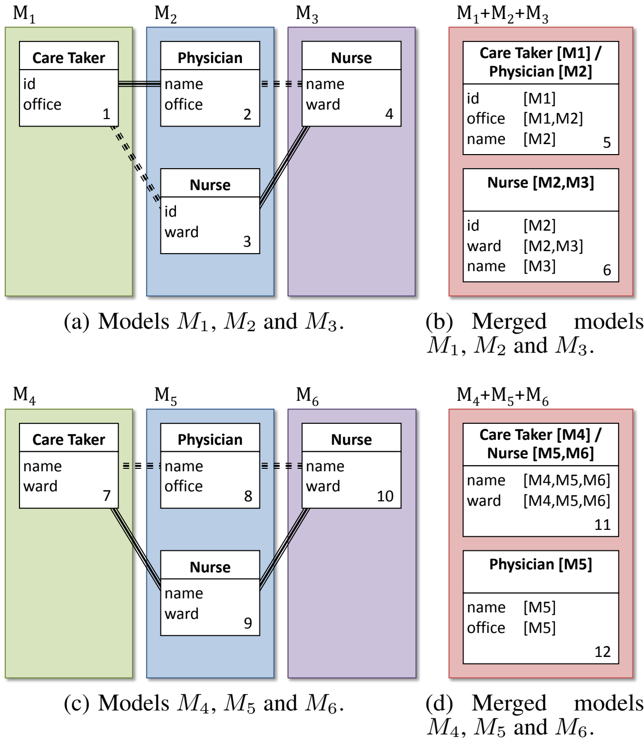
In this paper, we take a closer look at the problem of n-way merging. We refine existing compare, match and compose definitions to consider *tuples* rather than *pairs* of elements from multiple distinct input models. We focus on the matching step – the most challenging among the three – and show that it can be reduced to the *weighted set packing* problem which is known to be NP-hard [2]. We study and compare the state-of-the-art approximations to that problem, both theoretically and empirically, exploring their applicability to merging real-life models of a reasonable size. We discover that the scalability of these approximation algorithms is limited to a small number of small models and thus that they cannot be directly applied to real-life cases of model merging.

Further, we investigate the quality of the subset-based incremental approaches and propose our own polynomial-time heuristic n-way merging algorithm, *NwM*, that considers all input models simultaneously rather than processing them in any specific order. We empirically evaluate the quality and the performance of *NwM* by comparing it to the subset-based approaches, using as subjects two real-life and 300 randomly generated models. We show that *NwM* outperforms the alternative approaches in the majority of the cases, without any significant performance penalty.

Contributions. This paper makes the following contributions:

1. We formally define the n-way model merging problem.
2. We provide a theoretical and an empirical evaluation of the state-of-the-art approximation algorithms for n-way matching, via the *weighted set packing* problem.
3. We describe a number of polynomial-time subset-based approaches for n-way matching, as well as contribute a novel heuristic algorithm, *NwM*, that simultaneously considers all n input models together.
4. We provide a theoretical and an empirical evaluation of the proposed algorithms, showing cases where *NwM* significantly outperforms the alternative approaches.

The rest of the paper is structured as follows. Section 2 describes



(a) Models M_1, M_2 and M_3.

(b) Merged models M_1, M_2 and M_3.

(c) Models M_4, M_5 and M_6.

(d) Merged models M_4, M_5 and M_6.

Figure 1: Example UML Models.

the running examples and illustrates n-way merging. Section 3 gives background on the classical merge problem. We define n-way merge in Section 4 and study the applicability of existing solutions to this problem, via reduction to the NP-complete weighted set problem, in Section 5. We describe polynomial-time n-way matching approaches, including the novel *NwM* algorithm, in Section 6. We evaluate these approaches theoretically and empirically in Section 7. We discuss related work in Section 8 and conclude in Section 9.

2. EXAMPLE

We illustrate the n-way model merging problem using small UML model fragments inspired by our health care case study (see Figure 1(a)). The first model fragment, M_1, contains a single UML class `CareTaker` (element #1) which has two attributes: `id` that uniquely identifies the caretaker person and `office` that specifies where the person can be found. The second fragment, M_2, also contains two classes: `Physician` (element #2) and `Nurse` (element #3). Both classes have two attributes that help to identify and locate the corresponding persons: `name` and `office` for `Physician` and `id` and `ward` for `Nurse`.

When merging these two fragments, element #1 from M_1 can be combined either with element #2 or #3 from M_2. In fact, we cannot tell at this point which combination is better – in both cases, the classes have one shared and one non-shared attribute. Thus, it is conceivable to combine elements #1 and #3.

The third model fragment, M_3, contains a single class `Nurse` (element #4) that has two attributes: `name` and `ward`. As it was already decided to combine elements #1 and #3 in the previous step, element #4 can be combined either with the element that corresponds to their union or with element #2. None of these results are desired though: when considering all three fragments together, it becomes apparent that nurses belong to wards while physicians have offices. Thus, it seems more appropriate to combine element #1 with #2, and #3 with #4, as shown by bold lines connecting these elements in Figure 1(a). Figure 1(b) shows the result of the corresponding merge, where elements are annotated by models from which they originate: elements #5 and #6 represent the merge of element #1 with #2 and #3 with #4, respectively.

This example illustrates the need and the value of considering multiple models *simultaneously*, as a decision made in a certain iteration of a pairwise merging approach can impede reaching the desired result in later iterations. It also illustrates the sensitivity of the pairwise approach to the order in which the input models are picked: in this case, considering inputs in a different order, e.g., merging M_1 and M_3 first, would produce the desired result.

Fixing a particular order cannot guarantee the desired result in the general case though, as will be shown by our larger case studies. Intuitively, optimal matches are spread "across" models, as schematically shown by four model fragments in Figure 2. Picking any two models without considering the global picture might result in "binding" elements of the two models with each other, instead of "keeping" some elements available for later, more desired, combinations.

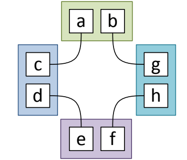


Figure 2: Optimal matches "across" models.

Figure 1(c) shows another set of model fragments inspired by the health care case study. Models M_4, M_5 and M_6 differ from the previous ones because the `CareTaker` class of M_4 (element #7) now has attributes `name` and `ward`, making it more similar to the `Nurse` class of M_5 (element #9) than to `Physician` (element #8). Also, the `Nurse` class of M_5 now has the attribute `name` instead of `id`. As the result, elements #7, #9 and #10 are combined together, while element #8 is just copied over to the merged result, as shown in Figure 1(d).

This example illustrates the case where the desired result is obtained by combining elements from three distinct models instead of producing pairs, like in Figure 1(b). Throughout the rest of the paper, we use the above examples to showcase the merging approaches being discussed.

3. BACKGROUND: MODEL MERGING

In this section, we review definitions for merging of two input models, following the presentation in [18] which divided the process into three steps: *compare*, *match* and *compose*.

$compare : M_1 \times M_2 \rightarrow [0..1]$ 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. Numerous specific implementations, analyzing structural and behavioral properties of the compared elements, exist. Most of these [21, 13, 16] calculate the similarity degree between two elements by comparing their corresponding sub-elements and weighing the results using empirically determined *weights*. 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 [16] also utilize behavioral properties of the compared elements, e.g., dynamic behaviors of states in the compared state machines, similar to checking bisimilarity.

$match : M_1 \times M_2 \times [0..1] \rightarrow M_1 \times M_2$ 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. Most implementations of *match* use empirically assigned *similarity thresholds* to decide such similarity. More advanced approaches, e.g., [6], rely on bipartite graph matching [15] (a.k.a. *the Hungarian algorithm*) to determine corresponding elements.

Finally, *compose* : $M_1, M_2, M_1 \times M_2 \rightarrow M$ 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. Specific compose algorithms define how to treat the matched (overlapping) and unmatched (non-overlapping) elements [4]. For example, the *union-merge* approach [20] assumes that matched elements are *complementary* and should be unified in the produced result, while unmatched elements are copied to the result without change. It is also possible for the composition to include only the overlap. Another option is to annotate the union-merge with origins of each element, as we did in Figures 1(b) and 1(d) by adding annotations like "[M1]" to indicate that the attribute `id` came from model M_1 (a.k.a. the *annotative software product line approach* [12, 18]).

4. N-WAY MODEL MERGING

In this section, we generalize the definition of merging given in Section 3 to n models. We formally specify the n-way counterparts of the *compare*, *match* and *compose* operators and discuss issues related to their implementation.

We assume n input models M_i, $i \in [0..n]$, of size k_i each. We use k to denote the size of the largest input model. Each model M_i contains uniquely identifiable elements $e_1 \ldots e_{k_i}$. Elements of distinct models form *n-tuples*. For an n-tuple t, we denote by $\mu(t)$ the ordered list of input models from which the elements of t originate.

DEFINITION 1. *An n-tuple (a.k.a.* a tuple*) t is a set of elements satisfying the following properties:*

(a) It contains at least one element: $|t| \geq 1$.

(b) No two elements belong to the same input model: $|t| = |\mu(t)|$.

For the example in Figure 1(a), a possible tuple can consist of elements #1, #2 and #4 from models M_1, M_2 and M_3, respectively, and is denoted by ⟨1,2,4⟩. μ(⟨1,2,4⟩)= $\{M_1, M_2, M_3\}$. Other possible tuples are ⟨1,2⟩ and ⟨1,3,4⟩ but not ⟨1,2,3⟩ since it contains two elements, #2 and #3, from model M_2.

In what follows, let T denote the set of all valid tuples for the input models M_i, $i \in [0..n]$. The size of T is $(\prod_{i=1}^{n}(k_i + 1)) - 1$, accounting for choosing an element from each model, including none, but disallowing an empty tuple.

4.1 Definition of the Operators

Compare assigns a similarity measure to a given tuple $t \in T$. We refer to the result of this function as a tuple's *weight* and denote it by $w(t)$. Thus, *compare* is a function that receives a tuple $t \in T$ and returns the similarity measure $w(t) \in [0..1]$ for its elements. The larger the value of w, 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. A *validity* function $\mathbb{V}$ decides whether a tuple is eligible to be selected. It extends a simple threshold-based comparison described in Section 3 to include more sophisticated validity criteria.

The weight of the set of tuples $\hat{T} \subseteq T$ produced by *match* is defined as a sum of weight of all tuples in $\hat{T}$ and is denoted by $w(\hat{T})$: $w(\hat{T}) = \sum_{\hat{t} \in \hat{T}} w(\hat{t})$. The larger the value of $w(\hat{T})$, the better is the produced match. Further, *match* should produce a disjoint set of tuples – in this work, we assume that an element can only be matched with a single element of any given input model, leading to the following definition:

DEFINITION 2. *Let* $\mathbb{V}$ *be a boolean validity function. Then, match is a function that returns a set of matches* $\hat{T} \subseteq T$ *that satisfy the following properties:*

(a) All tuples are valid: $\forall \hat{t} \in \hat{T} \cdot \mathbb{V}(\hat{t}) = true$.

(b) All tuples are disjoint: $\forall \hat{t}, \hat{t}' \in \hat{T} \cdot \hat{t} \cap \hat{t}' = \emptyset$.

(c) The set $\hat{T}$ *is maximal: no additional tuples can be added to* $\hat{T}$ *without violating constraints (a) and (b).*

For the example in Figure 1(a), *match* can output the tuples ⟨1,2⟩ and ⟨3,4⟩. It can also output either ⟨1,2,4⟩ or ⟨1,3,4⟩, but not both, since otherwise elements #1 and #4 would be matched to both elements #2 and #3 of M_2. Tuple ⟨1,3⟩ only is not a valid result as it can be augmented with the tuple ⟨2,4⟩.

Compose combines elements of each matched tuples in a particular way – any of the approaches discussed in Section 3 can be applied. Figure 1(b) shows a possible result of merging the input models in Figure 1(a), when *compose* assumes annotative software product line union-merge semantics and *match* produces tuples ⟨1,2⟩ and ⟨3,4⟩.

4.2 Towards Implementation of the Operators

Compare and *compose* usually encapsulate domain-specific information, i.e., which elements of a specific domain are considered similar and how such similar elements should be combined. Numerous works, e.g., [21, 13, 16, 1, 19], proposed implementations of these operators for both models and code, taking into account syntactical properties of the considered artifacts. On the other hand, *match* relies on the result of *compare* rather than domain-specific knowledge and is the least explored operator, especially for a collection of n inputs. Thus, the main focus of this paper is on the implementation of the n-way *match* step.

Yet, while a discussion of different ways for performing *compare* or *compose* is out of scope of this paper, we need to pick an operational definition of these operators which we describe below. We assume that each model element contains a set of typed properties, and we compare elements based on these. If model elements are UML classes, as in the example in Figure 1(a), possible properties can be class names, attributes, methods, associations to other classes, generalization relationships, etc. For example, element #1 in Figure 1(a) is defined by properties *id* and *office* of type UML class attribute and *Care Taker* of type UML class name. For simplicity, in our presentation we do not consider elements of types other than UML classes, although it is easy to extend the element / property approach to other types as well.

For an element e, we denote by $\pi(e)$ the set of all of its properties. Similarly, for a tuple $t = \langle e_1, \ldots, e_m \rangle$, we denote by $\pi(t)$ the set of distinct properties of all elements of t: $\pi(t) = \bigcup_{e_i \in t} \pi(e_i)$.

We assume, without loss of generality, that the goal of *compare* is to assign high weight to tuples whose elements share similar properties. For each tuple t and a property p, *compare* considers the number of elements in t that have that property. Then, for each tuple t, it calculates the distribution of properties: the number of properties that appear in j elements of t. We denote this number by n_j^p. For the example in Figure 1(a), the property `name` appeared twice in tuple ⟨1,2,4⟩ (in classes `Physician` and `Nurse`), as well as the property `office` (in classes `CareTaker` and `Physician`). The remaining properties, namely, attributes `id` and `ward`, as well as class names `CareTaker` and `Physician` and `Nurse` are unique to a single element in the tuple. Such properties "differentiate" tuple elements from each other and *compare* should "penalize" for that.

We thus define the *compare* function to assign a high weight to tuples with a large number of properties shared by a large number of elements: $w(t) = \frac{\sum_{2 \leq j \leq m} j^2 * n_j^p}{n^2 * |\pi(t)|}$.

The result is normalized by the number of input models n^2 (rather than the size of the tuple m^2) so that tuples with elements from a small subset of input models receive lower scores.

For example, the result of applying *compare* on the tuple ⟨1,2⟩ in Figure 1(a) is $\frac{2^2*1}{3^2*5} = \frac{4}{45}$: only the `office` attribute appears in both elements of the tuple, while there are five distinct properties in total – the class names `CareTaker` and `Physician`, and the class attributes `id`, `office` and `name`. Applying *compare* on tuples ⟨1,3⟩ and ⟨2,4⟩ results in $\frac{2^2*1}{3^2*5} = \frac{4}{45}$ as well. *Compare* on tuple ⟨3,4⟩ yields $\frac{2^2*2}{3^2*4} = \frac{2}{9}$; applying it to ⟨1,3,4⟩ also results in the same value, i.e., $\frac{2^2*3}{3^2*6} = \frac{2}{9}$. The weight of a tuple containing a single element, e.g., ⟨1⟩, as well as the weight of a tuple whose elements are completely disjoint, e.g., ⟨1,4⟩, is defined to be zero.

Our *validity* criteria require that each element of a tuple share at least one property with other elements. For example, the tuple ⟨1,4⟩ in Figure 1(a) is invalid, as classes `CareTaker` and `Nurse` have no attributes in common. Indeed, such classes should never be matched[1]. All of the remaining tuples in this example are valid.

Given the *compare* function and the validity criteria, the match with the maximal weight is produced by a combination of two tuples: ⟨1,2⟩ and ⟨3,4⟩. The weight of this solution is $\frac{4}{45} + \frac{2}{9} = \frac{14}{45}$, which is higher than picking the tuples ⟨1,3⟩ and ⟨2,4⟩ instead or a single tuple that combines three elements from distinct models: either ⟨1,2,4⟩ or ⟨1,2,3⟩. For the example in Figure 1(c), the best match corresponds to the solution that has a single tuple ⟨7,9,10⟩ with the weight $\frac{3^2*2+2^2*1}{3^2*4} = \frac{11}{18}$. In this case, the value is higher than for any other combination of tuples.

For *compose*, we assume the annotative SPL union-merge semantics, aligned with our broader research agenda. Its implementation is straightforward: elements of each tuple in the solution found by *match* are merged by unifying their properties, and are annotated by the corresponding source model for traceability purposes. Elements of the input models that are not part of any tuple produced by *match* are copied to the result as is. For the example in Figure 1(a), where *match* produced two tuples ⟨1,2⟩ and ⟨3,4⟩, the two elements in the merged model in Figure 1(b). i.e., elements #5 and #6, correspond to the merge of elements in the first and the second tuple, respectively. The merge of the models in Figure 1(c) is shown in Figure 1(d). Here, element #11 corresponds to the single tuple ⟨7,9,10⟩ returned by *match*. Element #12 corresponds to element #8 of the input model M_5 which is not part of any tuple and thus is copied to the result as is.

5. N-WAY MATCHING VIA WEIGHTED SET PACKING

In this section, we show that the n-way matching problem is reducible to the well-known NP-hard problem of *weighted set packing* [2]. We then consider the applicability of the existing approximations of this problem to merging software models.

5.1 Weighted Set Packing and Its Approximation Algorithms

The reduction to the *weighted set packing* problem easily follows from its definition: given a collection of weighted sets of cardinality at most n (in our case, approximately k^n tuples consisting of elements from n input models together with their weights as calculated by *compare*), the *weighted set packing* produces a collection of disjoint sets with the maximum total weight (in our case, the match).

[1]For tuples of size two, the similarity measure of invalid tuples is 0, but that does not hold for larger tuples.

The problem is NP-hard [2], and thus no solution that is polynomial in the size of the input (in our case, the set of tuples), exists.

There are a number of theoretically bounded approximations to that problem, polynomial in k^n (and thus exponential in n). Their main properties, i.e., approximation factor and time complexity, are summarized in the first four rows of Table 1. The simplest approximation algorithm (*Greedy*) [5] picks tuples with the maximal weight out of the set of all possible tuples, disregarding tuples that contain the elements already picked in the previous iterations. This algorithm can clearly miss the optimal result: picking a tuple with the maximal weight without "looking ahead" can block selection of combinations with a higher total weight. Since a selection of a tuple t with the weight w can block the selection of up to n other tuples whose weight cannot exceed w, the approximation factor of *Greedy* is n [5], i.e., the weight that this algorithm computes is within n times of the optimal. For the example in Figure 1(a), *Greedy* might pick the tuple ⟨1,3,4⟩ with the weight $\frac{2}{9}$, preventing the generation of the optimal solution with tuples ⟨1,2⟩ and ⟨3,4⟩ and weight $\frac{14}{45}$. For the example in Figure 1(c), *Greedy* does find the optimal solution – the tuple ⟨7,9,10⟩.

A number of approaches improve on *Greedy* by combining it with different types of local search [2, 5, 3]. They start from the solution found by *Greedy* and iteratively attempt to improve it by selecting s disjoint tuples that are not part of the current solution and trying to swap them with a minimal set of tuples in the solution so that tuples in the solution remain disjoint. The algorithms vary by the selection criterion for swapping (either the total weight of the solution increases or the square of the weight increases), the selection of the swapping candidate (either the first set that satisfies the swapping criterion or the best one) and by the size s of the candidate set. For the example in Figure 1(a), if *Greedy* produced the solution consisting of ⟨1,3,4⟩, such approaches can swap it with the pair of tuples ⟨1,2⟩ and ⟨3,4⟩, producing the optimal result. The approximation factors of these algorithms, as given by their corresponding authors [2, 5, 3], are summarized in Table 1.

Both *Greedy* and local search-based algorithms have high runtime and space complexity. In fact, even keeping all tuples in memory, let alone the information about which tuples are disjoint, is not possible for more than a few models with only around 30 elements each. In the last two columns of Table 1, we list time complexity for two version of each algorithm – the ones that keep all data in memory and the ones that do not.

5.2 Preliminary Evaluation

The time complexity estimations given in Table 1 might be too high for real-life scenarios. Also, our weight and validity functions reduce the space of tuples considered by the algorithms by assigning zero weight to a large set of tuples. We thus experimented with applying the proposed approximation solutions to the n-way matching problem on real-life examples.

We implemented the algorithms in Java, introducing numerous optimizations, including multi-threading, to improve execution times. We executed these algorithms on an Intel Core2Quad CPU 2.33GHz machine using JVM version 7.

As subjects, we picked two sets of models reported in [17]. The first consists of eight different variants of a Hospital system modeled as UML class diagrams. The system handles services within the hospital and describes the role of the hospital staff, its wards, different types of patients, and more. A small snippet of the models is shown in Figure 1(a). In total, there are 221 classes in all eight models, with an average of 27.63 classes per model. The largest model has 38 classes while the smallest has 18. Classes have 4.76 attributes on average (including name, associations, inheritance rela-

Table 1: Matching algorithms.

	Approximation factor	Time complexity	
		In memory	Memory efficient
Greedy (*Gr*)	n	$O(k^{2*n})$	$O(n^3 * k^{2n+1})$
Local Search Pick First	$n-1+1/s$	$O(k^{n*s+2*n+1} * s * n)$	$O(k^{n*s+2*n+1} * s * n^3)$
Local Search Pick Best	$2*(n+1)/3$	$O(k^{(n+1)^2} * n^3)$	$O(k^{(n+1)^2} * n^5)$
Local Search Squared Pick Best	$(n+1)/2$	$O(k^{(n+1)^2+n} * n^4)$	$O(k^{(n+1)^2+n} * n^6)$
Pairwise (*PW*)	–	$O(n * k^3)$	
Greedy 3 (*Gr3*)	–	$O(n * n^3 * k^7)$	
Greedy 4 (*Gr4*)	–	$O(n * n^3 * k^9)$	
NwM	–	$O(n^4 * k^4)$	

Table 2: Execution time for *Greedy* on Hospital and Warehouse cases.

Hospital				**Warehouse**			
k_1=26, k_2=28, k_3=38, k_4=25, k_5=22, k_6=18				k_1=18, k_2=31, k_3=27, k_4=24, k_5=25, k_6=19			
n=3	n=4	n=5	n=6	n=3	n=4	n=5	n=6
< 5s	1.86m	38.44m	10.9h	< 2s	46s	20m	5.9h

tionships, etc.). The largest class has 15 attributes. Around 30% of the attributes appear in more than 5 models.

The second set consists of sixteen different variants of a Warehouse system, also modeled as UML class diagrams. The system is designed to handle orders, track their status, provide computer support for warehousing workers, keep track of inventory levels in the warehouse, etc. Warehouse models have 388 elements in total, with an average of 24.25 classes per model. The largest and the smallest models have 44 and 15 classes, respectively. Classes have 3.67 attributes on average, with 11 attributes in the largest class. Around 15% of the attributes appear in more than 5 models. A complete description of both case studies can be found in [17].

Our experiment showed that none of the algorithms scaled to operate on the complete set of the input models, i.e., none achieved termination after 5 hours. Instead, we tried running the algorithms on the first three, four, five and six models from each of the case studies, in the order of their appearance in [17]. Algorithms based on local search failed to terminate after 5 hours even on a three-model subset. Execution times for *Greedy* are shown in Table 2. k_is capture the number of elements in each model.

Even though it might be possible to come up with more efficient implementations of the algorithms than ours, the results indicate that, generally, the algorithms do not scale well. *Greedy* seems to be the only feasible approach, and only for merging up to five small models (with 20-40 elements). This calls for a different solution for the n-way merging problem, polynomial in both k and n. We explore it in the remainder of the paper.

6. POLYNOMIAL-TIME APPROACH TO N-WAY MATCHING

In this section, we discuss algorithms which are polynomial both in the number of input models and in their size. First, we define a set of solutions that incrementally combine all input models in small subsets and discuss their properties (Section 6.1). We then present a novel algorithm, *NwM*, that considers all input models simultaneously (Section 6.2). Unlike the approximation algorithms for the weighted set packing problem, the algorithms presented in this section are not guaranteed to produce an answer within a particular factor of the optimal. We empirically evaluate the algorithms in terms of their quality and scalability in Section 7.

6.1 Subset-Based Approaches

A straightforward solution for merging n input models is to do so in smaller subsets, e.g., in pairs, performing $n-1$ pairwise combinations. To do so, we maintain a pool of models, with all input models initially in it. The algorithm iteratively selects and removes a subset of models from the pool, merges them together and puts the result back into the pool for further merging with additional models.

Subsets of size two can be merged using the bipartite graph matching algorithm [15] which produces a disjoint set of matches with the maximal total weight. The algorithm is based on combinatorial optimization techniques and solves the matching problem in time polynomial in size of the larger input model, returning an optimal result [8]. Larger subsets, of three or four models, can be merged using the *Greedy* algorithm described in Section 5.1. Applying *Greedy* on more than four models, or applying additional, more sophisticated, algorithms does not scale well, as shown in Section 5.2.

We thus define three subset-based algorithms: *PW* – pairwise matching, *Gr3* – *Greedy* on subsets of size 3 and *Gr4* – *Greedy* on subsets of size 4. These are summarized in Table 1. The quality of the result produced by these algorithms, in terms of the total weight of the solution, largely depends on the order in which the models are picked. For the example in Figure 1(a), selecting models M_1 and M_2 first can result in merging elements #1 and #3 with each other, as this combination is indistinguishable from the more desired combination of #1 and #2 since both pairs of elements have the same weight. As the result of this selection, it is impossible to generate the merged model shown in Figure 1(b) where element #1 is combined with #2, while #3 is combined with #4. Picking models M_1 and M_3 first could produce a better result if the highly dissimilar elements #1 and #4 are not merged. Then, during the next iteration, these elements could be combined with elements #3 and #2 from M_2, respectively, producing the desired combination. The

above discussion also makes it clear that subset-based incremental algorithms have no theoretical approximation factor since any merge of two elements produced in a given iteration can prevent future, more efficient, combinations.

To consider different orderings of input modes, we define and evaluate three variants of each subset-based algorithm. The first picks and processes the input models in the order of their appearance (i.e., at random). We denote such algorithms by *PW*, *Gr3* and *Gr4*. The second variant arranges the input models by size in ascending order, with the corresponding algorithms denoted by *PW*↑, *Gr3*↑ and *Gr4*↑. The third variant arranges them in descending order, with the corresponding algorithms denoted by *PW*↓, *Gr3*↓ and *Gr4*↓. For the example in Figure 1(a), algorithm *PW* picks models M_1 and M_2 first, *PW*↑ picks models M_1 and M_3, while *PW*↓ picks either M_2 and M_3, or M_2 and M_1. *Gr3*, *Gr3*↑ and *Gr3*↓ are equivalent in this case, as there are only three input models in total, so the ordering does not make any difference (and *Gr4* algorithms are not applicable at all).

We evaluate the relative effectiveness of these algorithms in Section 7. We also experimented with the idea of ordering input models by their cohesiveness, i.e., first merging those that are most similar, but observed that there is a strong correlation between this approach and the size-based ordering: larger models produce more matches which increases the total weight of the overall result and thus the similarity of these models.

6.2 The Algorithm NwM

In this section, we present a novel algorithm for n-way merging, *NwM*, which considers all n input models together and thus does not depend on any particular order of model selection (see Algorithm 1). Its main idea is based on picking optimal matches from *distinct* models and incrementally grouping them until a maximal set of tuples is produced. The algorithm obtains as input a set of tuples $\hat{T} \in T$ and outputs either a set of tuples $\hat{S} \in T$ over the same elements, with improved total weight: $w(\hat{S}) > w(\hat{T})$, or the input set $\hat{T}$, if such improvement is not possible.

Algorithm 1 *NwM*($\hat{T} \in T$): $\hat{S} \in T, w(\hat{S}) \geq w(\hat{T})$

1: **loop**
2: **for all** $\hat{t}_1, \hat{t}_2 \in \hat{T}$ **do** — Assign weights to pairs of input tuples
3: $w(\hat{t}_1, \hat{t}_2) \leftarrow \begin{cases} 0 & \neg\mathbb{V}(\hat{t}_1 \cup \hat{t}_2) \\ 0 & w(\hat{t}_1 \cup \hat{t}_2) < w(\hat{t}_1) + w(\hat{t}_2) \\ 0 & \mu(\hat{t}_1) \cap \mu(\hat{t}_2) \neq \emptyset \\ 0 & \hat{t}_2 <_{\mathbb{O}} \hat{t}_1 \\ w(\hat{t}_1 \cup \hat{t}_2) & otherwise \end{cases}$
4: **end for**
5: $\hat{P} \leftarrow$ *match*$(\hat{T}, \hat{T}, w)$
6: $\hat{C} \leftarrow \emptyset$
7: **while** $(|\hat{P}| > 0)$ **do** — Proceed with chaining for all matches
8: pick first $(\hat{t}, \hat{t}') \in \hat{P}$
9: $\hat{P} \leftarrow \hat{P} \backslash (\hat{t}, \hat{t}')$
10: **if** $(\exists c \in \hat{C} \mid c = [c_1, \ldots, c_n] \wedge c_n = \hat{t})$ **then** — Can append $\hat{t}'$?
11: **if** $(\mathbb{V}([c, \hat{t}']) \wedge \forall c_i \in \hat{C}(\mu(c_i) \cap \mu(\hat{t}') = \emptyset))$ **then** — if valid
12: $c \leftarrow [c, \hat{t}']$
13: **end if**
14: **else if** $(\exists c \in \hat{C} \mid c = [c_1, \ldots, c_n] \wedge c_1 = \hat{t}')$ **then** — Can prepend $\hat{t}$?
15: **if** $(\mathbb{V}([\hat{t}, c]) \wedge \forall c_i \in \hat{C}(\mu(c_i) \cap \mu(\hat{t}) = \emptyset))$ **then** — if valid
16: $c \leftarrow [\hat{t}, c]$
17: **end if**
18: **else**
19: $c \leftarrow [\hat{t}, \hat{t}']$ — Start a new chain
20: $\hat{C} \leftarrow \hat{C} \cup \{c\}$
21: **end if**
22: **end while**
23: $\hat{S} \leftarrow$ *optimize*$(\hat{C})$
24: **if** $(w(\hat{S}) = w(\hat{T}))$ **then** — No further improvements?
25: **return** $\hat{S}$
26: **else**
27: $\hat{T} \leftarrow \hat{S}$ — Reiterate
28: **end if**
29: **end loop**

Brief overview of the algorithm. In the first iteration, elements of all input models $M_1 \ldots M_n$ are represented by individual, single-element, tuples. For the example in Figure 1(a), the tuples are: $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3 \rangle$ and $\langle 4 \rangle$. Pairs of input tuples are assigned weights (Line 3) and matched using the bipartite graph *match* algorithm (Line 5). For each input tuple, the algorithm selects either zero or one matching counterpart, maximizing the total weight of the result. For the example in Figure 1(a), tuples $\langle 1 \rangle$ and $\langle 2 \rangle$, as well as $\langle 3 \rangle$ and $\langle 4 \rangle$, are matched with each other after the first iteration of the algorithm.

Matched tuples are further processed and unified (Lines 6-22), producing tuples $\langle 1,2 \rangle$ and $\langle 3,4 \rangle$ in the above example. The unified tuples are used as input to the next iteration of the algorithm (Line 27). The algorithm terminates when no matches of the input tuples can be made and thus no improvements can be achieved (Lines 24-25).

Detailed description of the algorithm. In every iteration, all pairs of input tuples are assigned weights. These weights are further used by the bipartite graph matching algorithm to produce an optimal match – a disjoint set of tuple pairs with the maximal total weight. Since matched tuples are further unified, we assign a pair of tuples $(\hat{t}_1, \hat{t}_2)$ the weight of the tuple that corresponds to their union (see the *otherwise* clause in Line 3 of Algorithm 1): $w(\hat{t}_1, \hat{t}_2) = w(\hat{t}_1 \cup \hat{t}_2)$. For the example in Figure 1(a), $w(\langle 1 \rangle,\langle 2 \rangle)$ = $w(\langle 1 \rangle,\langle 3 \rangle)$ = $w(\langle 2 \rangle,\langle 4 \rangle)$ = $\frac{4}{45}$ and $w(\langle 3 \rangle,\langle 4 \rangle)$ = $\frac{2}{9}$.

Some pairs correspond to tuples that should never be unified, e.g., when the unified tuple is invalid w.r.t. the validity function $\mathbb{V}$. For our example in Figure 1(a), the tuples $\langle 1 \rangle$ and $\langle 4 \rangle$ share no common properties and thus we treat their combination as invalid. While it is possible to filter such bad combinations after the matching is done, preventing their generation in the first place is preferred since the original tuples can then participate in more desired combinations. We thus assign weight 0 to combinations that are a priori "illegal", relying on the bipartite graph matching algorithm's ability to ignore pairs of elements with zero weight. Four types of such illegal combinations are described below (and encoded in Line 3 of Algorithm 1):

1. Pairs whose unification results in a tuple which is invalid w.r.t. the validity function $\mathbb{V}$, e.g., the pair ($\langle 1 \rangle$, $\langle 4 \rangle$).
2. Pairs for which the weight of the union is lower than the sum of input tuple weights – unifying such pairs is clearly not worthwhile. This situation cannot occur when unifying single-element tuples as their weight is zero and the unification can only increase it.
3. Pairs that contain elements from the same model, that is, $\mu(\hat{t}_1) \cap \mu(\hat{t}_2) \neq \emptyset$. For example, $\mu(\langle 1,2 \rangle)$ in Figure 1(a) is $\{M_1, M_2\}$, while $\mu(\langle 3,4 \rangle)$ is $\{M_2, M_3\}$. Unifying these tuples would result in a tuple with two elements, #2 and #3, from M_2, which is not allowed.
4. Pairs introducing "circular" dependencies between tuples, i.e., if a tuple $\hat{t}_1$ is matched with $\hat{t}_2$, $\hat{t}_2$ is matched with $\hat{t}_3$, and $\hat{t}_1$ and $\hat{t}_3$ contain elements from the same set of models, the unified tuple would be illegal. To limit circular dependencies, we introduce a partial order, $\mathbb{O}$, on the set of all tuples, such that $\hat{t}_1 <_{\mathbb{O}} \hat{t}_2$ iff $\mu(\hat{t}_1)$ is smaller than $\mu(\hat{t}_2)$ in the lexicographical order. For example, $\langle 1 \rangle <_{\mathbb{O}} \langle 1,2,4 \rangle <_{\mathbb{O}} \langle 2 \rangle$. A pair of tuples $(\hat{t}_1, \hat{t}_2)$ for which $\hat{t}_2 <_{\mathbb{O}} \hat{t}_1$ is assigned zero weight, e.g., the pair ($\langle 2 \rangle$, $\langle 1 \rangle$) but not the symmetric pair ($\langle 1 \rangle$, $\langle 2 \rangle$).

For the example in Figure 1(a), in the first iteration of the algorithm, only four pairs of tuples get a non-zero weight: ($\langle 1 \rangle$,$\langle 2 \rangle$), ($\langle 1 \rangle$,$\langle 3 \rangle$), ($\langle 2 \rangle$,$\langle 4 \rangle$), ($\langle 3 \rangle$,$\langle 4 \rangle$).

Results of *match* are represented by the map $\hat{P}$ which relates matched tuples to each other (Line 5). We say that $\hat{t}_1$ is matched with $\hat{t}_2$ if $\hat{P}(\hat{t}_1) = \hat{t}_2$. If a tuple $\hat{t}$ is not matched with any element, $\hat{P}(\hat{t})$ is *null*. The map is *ordered* by the weight of the matches, from the largest to the smallest, so that the "strongest" matches are retrieved first. For the example in Figure 1(a), two matches are produced in the first iteration of Algorithm 1: $\hat{P}(\langle 3 \rangle) = \langle 4 \rangle$ and $\hat{P}(\langle 1 \rangle) = \langle 2 \rangle$, with the weights $\frac{2}{9}$ and $\frac{4}{45}$, respectively. The example in Figure 1(c) also results in two matches: $\hat{P}(\langle 9 \rangle) = \langle 10 \rangle$ and $\hat{P}(\langle 7 \rangle) = \langle 9 \rangle$, with the weights $\frac{4}{9}$ and $\frac{2}{9}$, respectively.

Pairs of matched tuples are subject to internal processing. First, the tuples are "chained" towards possible unification that incorporates more than two tuples (Lines 10-17): if $\hat{P}(\hat{t}_1) = \hat{t}_2$ and $\hat{P}(\hat{t}_2) = \hat{t}_3$, the tuples are grouped into an ordered set c consisting of tuples $\hat{t}_1, \hat{t}_2, \hat{t}_3$. More generally, for the processed pair of tuples ($\hat{t}$, $\hat{t}'$) and the existing chain $c = [c_1, \ldots, c_n]$, if $\hat{t} = c_n$, the last tuple of the pair can be appended to the chain, producing the chain $[c_1, \ldots, c_n, \hat{t}']$ (Lines 10-13). Otherwise, if $\hat{t}' = c_1$, the first tuple of the pair can be prepended to the chain, producing the chain $[\hat{t}, c_1, \ldots, c_n]$ (Lines 14-17). For our example in Figure 1(c), the pair ($\langle 9 \rangle$, $\langle 10 \rangle$) is processed first; then the tuple $\langle 7 \rangle$ from the pair ($\langle 7 \rangle$, $\langle 9 \rangle$) is prepended to it, producing the chain $\langle 7 \rangle$, $\langle 9 \rangle$, $\langle 10 \rangle$. The chain corresponds to the tuple $\langle 7,9,10 \rangle$ which contains all three elements.

During chaining, the algorithm checks whether the appended / prepended tuple causes the union of the chained elements to be invalid w.r.t. the validity function $\mathbb{V}$. Similarly, it checks whether the tuple intersects with at least one other tuple of the chain w.r.t. their set of models: while we introduced a partial order on the set of tuples, for the pairs of matches $\hat{P}(\hat{t}_1) = \hat{t}_2$ and $\hat{P}(\hat{t}_2) = \hat{t}_3$, $\mu(\hat{t}_1) \cap \mu(\hat{t}_3)$ might not be empty, even though $\mu(\hat{t}_1) \cap \mu(\hat{t}_2) = \emptyset$ and $\mu(\hat{t}_2) \cap \mu(\hat{t}_3) = \emptyset$. In such cases, the tuple should not be added to the chain (Lines 11 and 15).

When both tuples of the matched pair do not belong to any chain, a new chain is started (Lines 18-21). That is the case for the first pair of tuples ($\langle 9 \rangle$, $\langle 10 \rangle$) in the example in Figure 1(c), as well as for both pairs of tuples in the example in Figure 1(a), ($\langle 1 \rangle$,$\langle 2 \rangle$) and ($\langle 3 \rangle$, $\langle 4 \rangle$).

Every tuple added to a chain is removed from $\hat{P}$ (Line 9); the chaining process continues until all tuples in $\hat{P}$ are processed (Line 7).

The chaining phase is followed by the optimization phase (Line 23), in which we check whether the chaining was inefficient, i.e., that it chained "too much" and splitting a chain c into smaller "sub-chains" $c_1 \ldots c_p$ improves the weight of the result: $w(c) < \sum_{1 \leq i \leq p} w(c_i)$. During the optimization step, we only verify whether the chain can be broken into one or more parts, without trying to reshuffle tuples of the chain and check their possible combinations. This heuristic is reasonable as the chains are built while putting optimally matched tuples close to each other.

Optimized chains form tuples that are used as input to the next iteration of the algorithm (Line 27) and the process continues until no further improvements can be achieved. For the example in Figure 1(a), the algorithm stops after the first iteration, producing tuples $\langle 1,2 \rangle$ and $\langle 3,4 \rangle$. No further unification is possible as the combination of these two tuples is invalid – it contains two elements from the same model M_2. The result produced by the algorithm in this case corresponds to the desired solution in Figure 1(b). Likewise, for the example in Figure 1(c), the algorithm stops after producing the chained tuple $\langle 7,9,10 \rangle$, which also corresponds to the desired solution in Figure 1(d).

Validity of the algorithm. By construction, the algorithm ensures generation of valid matches, per Definition 2:

(a) *All tuples are valid*: Pairs of tuples that share elements from the same model are assigned zero weight and thus cannot be formed (see Line 3), while the chaining step ensures that chains do not contain elements from the same model (see Lines 11 and 15). Likewise, pairs of tuples that are invalid w.r.t. the validity function $\mathbb{V}$ are assigned zero weight and thus cannot be formed (see Line 3), while both the chaining and the optimization steps ensure validity of tuples that correspond to the produced chains (see Lines 11 and 15).

(b) *All tuples are disjoint*: In the first iteration, all input model elements belong to separate tuples. In subsequent iterations, a tuple can be added to one chain only (see Lines 9-19) and thus becomes part of only one tuple. By induction, an element cannot belong to more than one output tuple.

(c) *The set $\hat{T}$ is maximal*: The algorithm starts with the set of all input model elements, thus all elements are part of at least one tuple. It is not possible to expand the solution without violating the above disjointness constraint (b).

Clearly, the algorithm is heuristic and does not produce optimal results in all cases. Yet, it is reasonable to believe that it has a high chance of producing good matches because it simultaneously considers best combinations across different models, rather than limiting itself to any particular ordering. We evaluate the algorithm and compare it to subset-based approaches in the next section.

7. EVALUATION

In this section, we discuss runtime and space complexity of the polynomial algorithms described in Section 6 and then report on an empirical evaluation of the effectiveness of these approaches.

7.1 Theoretical Evaluation

As can be seen in Table 1, the runtime complexity of our *NwM* algorithm is bounded by $O(n^4 * k^4)$, making the algorithm polynomial in both n (the number of input models) and k (the size of the largest input model): there are a total of $n * k$ elements considered by the algorithm and each iteration reduces the number of input tuples by at least one. Bipartite graph matching of $n * k$ elements in our implementation is bounded by $(n * k)^3$ [15] (even though more efficient implementations are also possible [8]); the chaining phase is quadratic in the size of the input; and the optimization phase is quadratic to the maximal length of the chain. Since all tuples in a chain contain elements from distinct models, the length of the chain is bounded by n. The space complexity of the algorithm is bounded by the maximal number of matched pairs, that is, $n^2 * k^2$.

The runtime complexity of all *PW* algorithms is $O(n * k^3)$: they perform up to n iterations, each of which takes $O(k^3)$. The space complexity of these algorithms is $O(k^2)$, making them more efficient than *NwM* both w.r.t. time and memory.

The runtime complexity of algorithms *Gr3* and *Gr4* is $O(n * n^3 * k^7)$ and $O(n * n^3 * k^9)$, respectively, with the space complexity of $O(1)$ for both cases (we implemented the memory efficient version of the algorithms). That is, the *Gr3* algorithms have similar complexity to *NwM*, while the *Gr4* algorithms are more expensive.

As discussed earlier and indicated in Table 1, all of the above algorithms do not have a theoretical approximation factor – any selection they make can result in matches that, while reasonable in the current iteration, "block" selected elements from participating in tuples with a higher weight.

Table 3: Weights and execution times of *NwM* compared to subset-based approaches. For each case, the results of *NwM* and the best subset-based approach are boldfaced.

		NwM	Subset-based approaches								
			PW	*PW↑*	*PW↓*	*Gr3*	*Gr3↑*	*Gr3↓*	*Gr4*	*Gr4↑*	*Gr4↓*
Hospital	Weight	**4.595**	4.473	4.475	4.514	4.268	4.017	4.453	4.356	4.382	**4.566**
	Time	**42.7s**	< 1s	< 1s	< 1s	16.1s	14.1s	19.6s	4.6m	4.2m	6.0m
Warehouse	Weight	**1.522**	0.973	0.981	**1.126**	1.018	1.044	1.037	1.013	1.111	1.000
	Time	**2.9m**	1.4s	1.2s	< 1s	45.4s	34.9s	42.8s	13.9m	27.9m	12.1m
Random models	Weight	**0.979**	0.835	0.756	0.842	0.849	0.795	0.857	**0.863**	0.836	0.853
	Time	**1.6m**	< 1s	< 1s	< 1s	19.5s	18.6s	23.7s	9.3m	12.8m	10.1m
"Loose" scenario	Weight	**0.980**	0.772	0.695	0.832	0.845	0.772	**0.855**	0.848	0.809	0.837
	Time	**1.5m**	< 1s	< 1s	< 1s	29.3s	21.9s	31.2s	13.1m	16.6m	12.2m
"Tight" scenario	Weight	**0.941**	0.947	0.900	0.954	0.930	0.897	0.940	0.941	0.920	**0.957**
	Time	**1.6m**	< 1s	< 1s	< 1s	23.7s	18.2s	25.3s	6.4m	6.9m	6.6m

7.2 Empirical Evaluation

We now report on an empirical evaluation of the polynomial-time n-way merging approaches described in Section 6. Specifically, our aim was to answer the following research questions:

RQ1. How does the performance of *NwM* compare to the subset-based approaches? What are the conditions under which each algorithm is better?

RQ2. How does the size of the selected subset affect the quality of the results produced by the subset-based approaches?

RQ3. How does the order in which input models are picked affect the quality of the results produced by these approaches?

Subjects. For our evaluation, we used the Hospital and Warehouse examples from [17], described in Section 5.2. Before merging, we performed a simple pre-processing step on the models, unifying syntactically similar properties, e.g., "patient" and "patients", since implementing *compare* algorithms is out of scope of this work. We augmented the set of subjects by 300 randomly generated models, divided into 30 sets of 10 models each. The first 10 sets of 10 models mimic the real-life examples in terms of the number of models and their sizes, the number of properties for each element, the total number of properties in all models and their distribution into shared and unshared (see Section 5.2). We refer to those as the *Random* case. Based on our experience with the Random case, and to gain deeper insights in the qualities of the evaluated algorithms, as well as the conditions that affect their performance, we generated two additional random cases, containing 10 sets of 10 models each. The cases, referred to as *the "Loose" scenario* and *the "Tight" scenario*, vary in the number of classes in each model, the total number of properties in all models, and the number of properties in each class, as discussed below. All models are available at `http://www.cs.toronto.edu/~mjulia/NwM`.

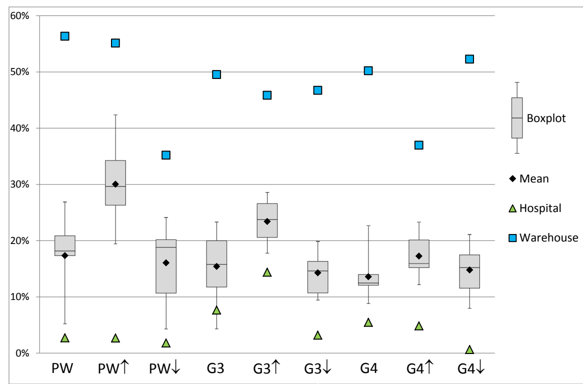
Methodology and Measures. We implemented all algorithms in Java and executed them on an Intel Core2Quad CPU 2.33GHz machine using JVM version 7. For fairness, we used the same implementation of *compare* and the same *validity* criteria for all algorithms (see Section 4.2): each element of a tuple in the solution must share at least one property with other elements of the tuple, and the more properties are shared the higher is the tuple weight.

For each algorithm, we measured the execution times and the weights of the produced solution for each of the case studies. We focused our evaluation on the matching stage rather than the perceived "appeal" of the result, which is largely domain-specific and depends on the definition of *compare*. Hence, we considered an algorithm to be better if it produces a solution with a higher total weight. We represented the result returned by each subset-based approach as a set of tuples, calculating the weight of each tuple and the weight of the overall solution as if they were produced by *NwM*.

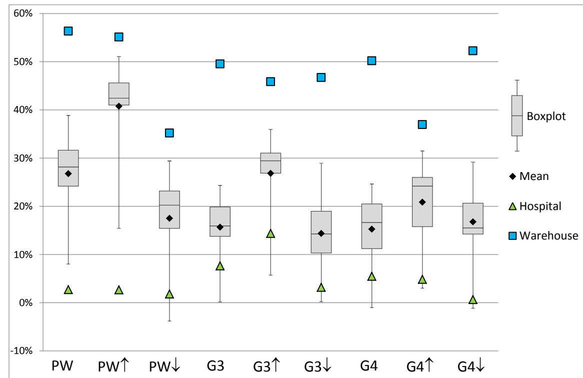
The results are summarized in Table 3. The first two rows show the results for the Hospital and Warehouse cases, while the other three show average results for the *Random*, the *"Loose" scenario* and the *"Tight" scenario*. For those cases, we merged each set of 10 models individually and averaged the results for all sets within a case. For comparison, we mark in bold the results produced by *NwM* as well as the best result achieved by any of the subset-based algorithms on the same case.

To compare *NwM* to each of the subset-based approaches further, we calculated the percentage by which it improved (or degraded) the weight of the solution, compared to the other algorithms. For example, in the Hospital case, *NwM* found a match with the total weight of 4.595, compared to 4.473 found by *PW*. This *improves* the matching by 2.7%. *Gr4↓* performed the best of the subset-based algorithms, achieving the result of 4.566. *NwM* improves that by only 0.6%. *Gr4↑* performed the worst, and *NwM* improves its result by 14%. For the Warehouse case, *NwM* finds a solution with the total weight of 1.522 which improves the best result of 1.126 found by *PW↓* by 35%.

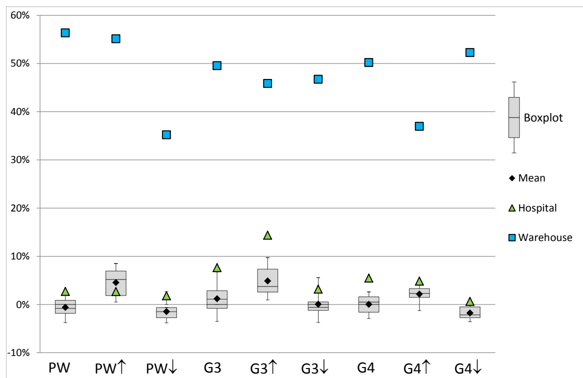
For the random cases, we calculated the percentage of weight increase / decrease of each run individually and summarized the distribution of results as boxplots in Figure 3, separately for each case. On the horizontal axis, we distinguish the nine algorithm variants used for comparison. On the vertical, we show the percentage of weight increase / decrease produced of our algorithm compared to each variant. For instance, compared to *PW*, the results achieved by *NwM* in the Random case (Figure 3(a)) range from 5.2% to 26.9% improvement (indicated by the thin black vertical line); half of the cases fall into a range between 17.3% to 20.9% improvement (indicated by the grey box showing the upper and lower quartile). The average improvement in this case is also 17.3% (indicated by a dia-



(a) Random case.



(b) "Loose" scenario case.



(c) "Tight" scenario case.

Figure 3: Results of comparing *NwM* against subset-based algorithms.

mond symbol) and the median is 18.2% (indicated by a horizontal line within the grey box). In addition, we also plot the results of the Hospital and the Warehouse cases (indicated by triangle and square symbols, respectively). The Hospital and the Warehouse results are also shown in Figure 3(b) and (c), together with the respective results for the "Loose" and the "Tight" scenarios.

Analysis and Results. The results show that for both the Hospital and Warehouse cases, *NwM* outperforms the existing approaches w.r.t. the weight of the found solution, i.e., the relative values are positive and appear above the 0% line in Figure 3. This result is also seen in the Random case (see Figure 3(a)): *NwM* is able to achieve 30% improvement on average compared to *PW*↑ and at least 13.5% on average compared to *Gr4*.

We also noticed that in the Warehouse case, *NwM* was able to achieve a much more substantial improvement than in the Hospital

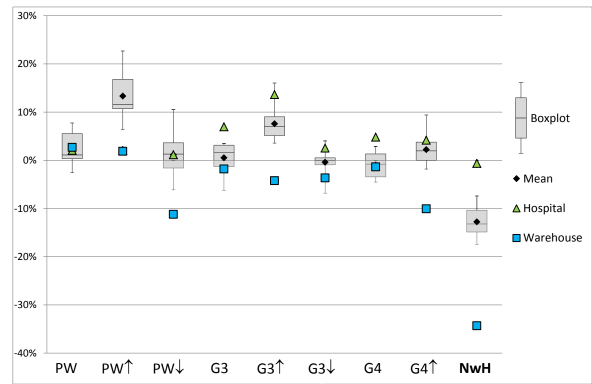


Figure 4: Relative performance of *Gr4*↓ on the Random case.

case, with Random being "in the middle". We analyzed the differences between these two cases in more detail. It appears they mostly differ in the total number of properties in all input models (159 properties in total in the Hospital example, compared to 338 for the Warehouse) and the average number of properties in each class (4.76 for the Hospital case and 3.67 for the Warehouse). Also, the range of model sizes differs from 18-38 classes in each model in the Hospital case to 15-44 classes in the Warehouse case. Thus, the Hospital case appears to be more "tight" – the model size distribution is smaller, there is a smaller number of properties in total, with more properties that appear in each class. The Warehouse case, on the other hand, is more "loose" w.r.t. these parameters.

We thus produced the "Loose" and "Tight" scenarios, varying the number of the above discussed parameters. Indeed, our algorithm was able to achieve much more substantial improvements in the "Loose" case (see Figure 3(b)): more than 40% on average compared to *PW*↑ and at least 14.3% on average compared to *Gr3*↓. In the "Tight" case (see Figure 3(c)), the best improvement of 4.8% on average was achieved compared to *Gr3*↑, while *Gr4*↓ outperforms our algorithm by 1.7%.

It is not surprising that subset-based approaches perform reasonably well in "tight" combinations, with *Gr4*↓ being the best in most cases: the merged elements are more similar to each other, thus allowing these algorithm to form valid matches in each step. For example, even if a subset-based algorithm makes a "wrong" selection, the remaining tuples still have a high enough weight as their elements are close to each other. For such cases, *NwM* performs similarly to the subset-based algorithms.

With respect to the execution time of the approaches, *PW* algorithms are faster than the others, terminating in less than one second (see Table 3), as expected. Execution times for *G3* ranged between 14 and 40 seconds, while for *NwM* they were 43 seconds to 2.9 minutes. *G4* was significantly slower than the other algorithms, taking from 4.2 minutes on the Hospital case to 27.9 minutes in the Warehouse case.

Conclusion – RQ1: Our experiments confirm that the *NwM* algorithm produces better results the subset-based approaches in the majority of cases, especially in the more "loose" combinations. This includes the real-life models that we used as input. The weight increases achieved by the algorithm are substantial while the decreases are rare and minor. Furthermore, the running time of *NwM* is feasible.

Comparing the subset-based approaches to each other, it appears that the *Greedy* approaches perform better than *Pairwise*. On average, *Gr4*↓ outperformed the other algorithms in the Hospital and "Tight" cases, while *Gr4* outperformed the others in the Random case. *Gr3*↓ was better than the alternative subset-based approaches

in the “Loose” case. Warehouse was the only case in which *PW*$\downarrow$ outperformed the rest of the subset-based algorithms.

To investigate whether it is beneficial to take a more “global” view, i.e., by combinig input models into larger subsets, we compared *Gr4*$\downarrow$ to the other approaches on the Hospital, Warehouse and Random cases, presenting the results in the boxplot view in Figure 4. The results appear inconclusive – most of the average improvements are only slightly higher than 0%, so there appears to be no benefit using this algorithm compared to the others. The figure also shows, again, that *NwM* performs significantly better than *Gr4*.

Conclusion – RQ2: Selecting larger subsets of input models, e.g., 4, to merge does not have a significant effect on the results.

For each of the subset-based approaches, we also evaluated the correlation between the quality of the produced match and the order in which the subset of input models is picked. As shown in Table 3, in all but two cases the strategy of combining models by size in the descending order, i.e., from the largest to the smallest, produced better results than those using the ascending order. Intuitively, we believe that handling larger models first produces a large model as a result, providing more “choices” for subsequent iterations which leads to better matches. While we saw some outliers, in the majority of cases, arranging models by size in the descending order was a more beneficial strategy.

Conclusion – RQ3: For subset-based approaches, the quality of the produced result is sensitive to the order of input models, with arranging them by size in the descending order being more beneficial.

7.3 Threats to Validity

Threats to external validity are most significant for our work. These arise when the observed results cannot generalize to other case studies. We attempted to mitigate these threats by using two real-life case studies and a considerable number of randomly generated models that mimic the properties of the real ones.

Using automatically generated models as a basis for evaluation is by itself a yet another threat to validity. We attempted to mitigate this by basing our generator on characteristics taken from the real-life models and using a combination of generated and real-life models for evaluation.

We experimented with a particular weight function, described in Section 4. Thus, the results might not generalize to other possible weight calculations. Yet, we believe that specific calculations of weights are orthogonal to our work, as they have a similar effect on all of the compared approaches.

8. RELATED WORK

Numerous approaches for model merging focus on merging two inputs with each other [21, 13, 16]. Similarly, model and code differencing approaches [11, 9, 1] focus on identifying differences in two, usually subsequent, versions of the input artifacts. Some works, e.g., [14], propose techniques for detecting many-to-many matches between the elements of two input models. Our work differs from those by addressing the problem of merging n input models together, for any $n > 2$.

Duszynski et al. [7] emphasize the need for simultaneous analysis of multiple source code variants, identify commonality and variability of those variants, and propose an approach for comprehensible visualization of the analysis results. This work does not attempt to study and optimize the matching step though, but rather greedily finds a match for each input element.

Approximation algorithms for the weighted set packing problem, applicable for the n-way matching step, are the most relevant to our work (see also Section 5.1). Arkin and Hassin [2] propose an algorithm based on local search. The algorithm starts from any solution, e.g., an empty set or a solution found by *Greedy* algorithm. It then iteratively attempts to improve the solution by selecting $2 \leq s \leq n$ disjoint tuples that are not part of the it, and trying the swap them with a subset of tuples in the solution, if that increases the solution's weight while still keeping it disjoint.

Chandra and Halldorsson [5] further improve that algorithm: instead of doing any local improvement that increases the weight of the solution, the authors propose to find and use the *best possible* improvement. The algorithm also assumes that all improvements are of size n, which makes it more expensive computation-wise.

Berman [3] proposes a yet another improvement to the above algorithm: instead of performing the swap with the best possible improvement, the algorithm performs a swap if the square of tuple weights is improved. Again, this increases the computational complexity of the algorithms.

The exact approximation factors and time complexity of the above algorithms are shown in Table 1. As discussed in Section 5.2, these algorithms do not scale for more than a small number of small models, while we aim to provide a practical solution that can be applied for merging real-life software models.

9. CONCLUSIONS AND FUTURE WORK

Merging multiple inputs together is an important task in several software development activities. These include combining overlapping views of different stakeholder or unifying multiple related products into a single-copy software product line representation. Yet, most of the existing works focus on merging two inputs together, providing little guidance on how to handle *multiple* inputs.

In this paper, we extended the model merging problem to consider n inputs. We focused on the most challenging step of the merging process – *matching* – and showed that the problem is NP-hard. We surveyed and evaluated state-of-the-art approximations developed in the literature, as well as current practices of incrementally merging n input models together in smaller subsets. Based on this experience, we proposed a novel, polynomial-time heuristic algorithm *NwM* that considers all n input models simultaneously. We evaluated our approach on a large set of cases, including two real-life examples, and showed that it achieves substantial improvements over approaches that merge input models in subsets, without a substantial degradation in performance. However, our approach, as well as other available scalable approaches, does not provide any theoretical approximation guarantees.

There are several directions for continuing this work. First, we are interested in exploring additional heuristics that could further improve the quality of the merge produced by *NwM* while keeping its execution time reasonable. For example, one could experiment with the idea of reshuffling chained tuples (rather than only breaking the chain into pieces) or even reshuffling elements from distinct tuples of the chain. We also plan to augment the work by integrating it with domain-specific *compare* and *compose* approaches and by extending it beyond numerical definition of matching quality, e.g., providing means to differentiate between *match* solutions with identical total weight. Further analyzing subset-based approaches, e.g., comparing all input models to each other and merging them in the order induced by an algorithm for finding a spanning tree, is a yet another direction for possible future work.

Acknowledgments. We thank Mark Braverman, Michael Kaminski and Dror Rawitz for many useful discussions during the course of this work. We also thank Magnús Halldórsson, Refael Hassin and Piotr Berman for helpful clarifications of the local search-based algorithms and the anonymous reviewers for their valuable comments.

10. REFERENCES

[1] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proc. of ESEC/FSE'11*, pages 190–200, 2011.

[2] E. M. Arkin and R. Hassin. On Local Search for Weighted k-Set Packing. *Mathematics of Operations Research*, 23(3):640–648, 1998.

[3] P. Berman. A d/2 Approximation for Maximum Weight Independent Set in d-Claw Free Graphs. In *Proc. of SWAT'00*, volume 1851 of *LNCS*, pages 214–219, 2000.

[4] 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.

[5] B. Chandra and M. M. Halldórsson. Greedy Local Improvement and Weighted Set Packing Approximation. *J. Algorithms*, 39(2):223–240, 2001.

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

[7] S. Duszynski, J. Knodel, and M. Becker. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Proc. of WCRE'11*, pages 303–307, 2011.

[8] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264, 1972.

[9] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE TSE*, 33:725–743, 2007.

[10] P. Frenzel, R. Koschke, A. P. J. Breu, and K. Angstmann. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. In *Proc. of WCRE'07*, pages 160–169, 2007.

[11] D. Jackson and D. A. Ladd. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In *Proc. of ICSM'94*, pages 243–252, 1994.

[12] 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.

[13] 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.

[14] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Gueheneuc. MADMatch: Many-to-many Approximate Diagram Matching for Design Comparison. *IEEE TSE*, 2013.

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

[16] 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.

[17] Y. T. Rad and R. Jabbari. Use of Global Consistency Checking for Exploring and Refining Relationships between Distributed Models: A Case Study. Master's thesis, Blekinge Institute of Technology, School of Computing, January 2012.

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

[19] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *Proc. of FASE'13*, pages 83–98, 2013.

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

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