Scalable N-Way Model Matching Using Multi-Dimensional Search Trees

Alexander Schultheiß*, Paul Maximilian Bittner§, Lars Grunske*, Thomas Thüm§ and Timo Kehrer*
*Humboldt University of Berlin, Germany
{schultha, grunske, kehrerti}@informatik.hu-berlin.de
§University of Ulm, Germany
{paul.bittner, thomas.thuem}@uni-ulm.de

***Abstract*—Model matching algorithms are used to identify common elements in input models, which is a fundamental precondition for many software engineering tasks, such as merging software variants or views. If there are multiple input models, an n-way matching algorithm that simultaneously processes all models typically produces better results than the sequential application of two-way matching algorithms. However, existing algorithms for n-way matching do not scale well, as the computational effort grows fast in the number of models and their size. We propose a scalable n-way model matching algorithm, which uses multi-dimensional search trees for efficiently finding suitable match candidates through range queries. We implemented our generic algorithm named RaQuN (Range Queries on N input models) in Java, and empirically evaluate the matching quality and runtime performance on several datasets of different origin and model type. Compared to the state-of-the-art, our experimental results show a performance improvement by an order of magnitude, while delivering matching results of better quality.**

***Index Terms*—Model-driven engineering, n-way model matching, clone-and-own development, software product lines, multi-view integration, variability mining.**

I. Introduction

Matching algorithms are an essential requirement for detecting common parts of development artifacts in many software engineering activities. In domains where model-driven development has been adopted in practice, such as automotive, avionics, and automation engineering, numerous model variants emerge from cloning existing models [1]–[3]. Integrating such autonomous variants into a centrally managed software product line in extractive software product-line engineering [4] requires to detect similarities and differences between them, which in turn requires to match the corresponding model elements of the variants. Moreover, matching algorithms are an indispensable basis for merging parallel lines of development [5], or for consolidating individual views to gain a unified perspective of a multi-view system specification [6].

Currently, almost all existing matching algorithms can only process *two* development artifacts [7]–[18], whereas the aforementioned activities typically require to identify corresponding elements in *multiple* (i.e., $n > 2$) input models. A few approaches calculate an n-way matching by repeated two-way matching of the input artifacts [19]–[24]. In each step, the resulting two-way correspondences are simply linked together to form *correspondence groups* or *matches* (aka. *tuples* [25]).

However, sequential two-way matching of models may yield sub-optimal or even incorrect results because not all input artifacts are considered at the same time [25]. The order in which input models are processed influences the quality of the matching because better match candidates may be found after an element has already been matched. An order might be determinable if a reference model is given, but this is typically not the case [6], [19], [26]–[30]. An optimal processing order cannot be anticipated and applying all $n!$ possible orders for n input models is clearly infeasible [21].

The only matching approach which simultaneously processes n input models is a heuristic algorithm called NwM by Rubin and Chechik [25]. NwM delivers n-way matchings of better quality than sequential two-way matching. Yet, we faced scalability problems when applying NwM to models of realistic size, comprising hundreds or even thousands of elements. The most likely reason for this is the huge number of model element comparisons, which often leads to performance problems even in the case of few input models if these models are large [31]–[33]. Thus, there is a strong need for a scalable n-way matching solution.

We propose RaQuN (**Ra**nge **Qu**eries on **N** input models), a generic, heuristic n-way model matching algorithm. The key idea behind RaQuN is to map the elements of all input models to points in a numerical vector space. RaQuN embeds a multi-dimensional search tree into this vector space to efficiently find nearest neighbors of elements, i.e., those elements which are most similar to a given element. By comparing an element only with its nearest neighbors serving as match candidates, RaQuN can reduce the number of required comparisons considerably.

For our empirical assessment, we use datasets from different domains and development scenarios. Next to academic and synthetic models [25], [34], we investigate variants generated from model-based product lines [35]–[38], and reverse-engineered models from clone-and-own development [1], [39]. Our evaluation shows that RaQuN reduces the number of required comparisons by more than 90% for most experimental subjects, making it possible to match models of realistic size simultaneously. In summary, our contributions are:

Generic Matching Algorithm (Section III). We present RaQuN, a generic simultaneous n-way model matching

This work has been supported by the German Research Foundation within the project *VariantSync* (KE 2267/1-1 and TH 2387/1-1).

algorithm using multi-dimensional search trees.

Domain-agnostic Configuration (Section IV). For all variation points of the generic algorithm, we propose domain-agnostic configuration options turning RaQuN into an off-the-shelf n-way model matcher.

Empirical Evaluation (Section V). We show that RaQuN has good scaling properties and can be applied to large models of various types, while delivering matches of better quality than other approaches.

II. N-Way Matching

In this section, we illustrate the n-way model matching problem with a simple running example, and discuss how algorithmic approaches calculate a matching in practice. The three UML class diagrams A, B, and C given in Fig. 1 are fragments of the hospital case study [25], [34]. Each of the three models is an early design variant of the data model of a medical information system. We use the symbolic identifiers 1 to 8 to uniquely refer to the models' classes.

Our representation of models follows the so-called *element-property approach* [25]. A model M of size m is a set of *elements* $\{e_1, \ldots, e_m\}$. Each model element $e \in M$, in turn, comprises a set of *properties*. For our running example, we consider UML classes as elements, and we restrict ourselves to two kinds of properties, namely class names and attributes. However, the element/property approach is general enough to account for other kinds of model elements (e.g., states and transitions in statecharts), and other kinds of properties (e.g., element references or element types).

Intuitively, n-way matching refers to the problem of identifying the common elements among a given set of n input models. A reasonable matching for our example is illustrated in Fig. 1, indicated by solid lines. The models A and B each contain a class named *Physician*. Both classes have several attributes in common and may thus be considered to represent "the same" conceptual model element in different variants. Following common terminology from the field of two-way matching, we say that class *Physician* in model A *corresponds to* class *Physician* in model B. Similarly, each of the three models contains a class named *AdminAssistant*, and all three variants of the class share several identical attributes. Thus, these classes form a so-called correspondence group (aka. *tuple* [25]). We call such a correspondence group a *match*.

Formally, we define an n-way matching algorithm as a function which takes as input a set $\mathcal{M} = \{M_1, ..., M_n\}$ of input models and returns a *matching* T. A matching $T = \{t_1, \ldots, t_k\}$ is defined as a set of matches, where each match $t \in T$ is a non-empty set of model elements. Analogously to all existing approaches to n-way matching [19]–[25], we assume matches in T to be mutually disjoint, and that no two elements of a match belong to the same input model. Formally, a match t is valid if it satisfies the condition

$$t \neq \emptyset \quad \wedge \quad |t| = |\mu(t)| \tag{1}$$

where $\mu(t)$ denotes the set of input models from which the elements of t originate. The intuitive matches illustrated in

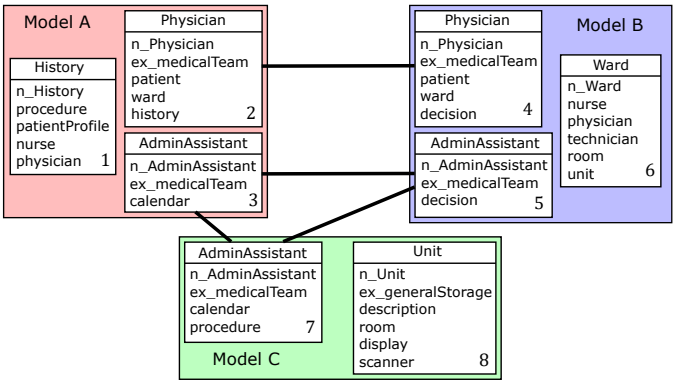

Fig. 1. Three UML models representing early design variants of the data model of a medical information system, serving as running example.

Fig. 1, i.e., $\{3, 5, 7\}$, $\{2, 4\}$, $\{1\}$, $\{6\}$, and $\{8\}$, are valid and mutually disjoint.

In theory, a matching could be computed by considering all possible matches for a set of input models. However, this approach is not feasible, as the number of possible matches for a set of models is equal to $(\prod_{i=1}^{n}(m_i + 1)) - 1$ [25], where n denotes the number of models, and m_i denotes the number of elements in the i-th model.

A trivial approach would be to rely on persistent identifiers or names of model elements. The limitations of such simple approaches have been extensively discussed in the literature on two-way matching [8], [9], [31]–[33] (cf. related work in Section VI), and also apply to the n-way model matching problem. Reliable identifiers are hardly available across sets of variants, and names are not sufficiently eligible for taking an informed matching decision without considering other properties. In particular, names are not necessarily unique, and some model elements do not have names at all [40].

In practice, matching algorithms thus operate heuristically. This requires a notion for the *quality* of a match, or in other words, a measure for the similarity of matched elements. Given a match $t \in T$, a *similarity function* calculates a value representing the similarity of the elements in t. We assume that a similarity function makes it possible (i) to establish a partial order on a set of matches, and (ii) to determine whether a set of candidate elements should be matched. An example for a similarity function is the weight metric introduced by Rubin and Chechik [25] (see Section IV-C).

III. Generic Matching Algorithm

In this section, we first describe our generic n-way matching algorithm RaQuN (Alg. 1), followed by an illustration applying the algorithm to our running example introduced in Section II, and closing with a theoretical analysis of the algorithm's runtime complexity. We focus on the high-level steps that are performed by the algorithm, while we discuss the details of how each step can be configured later in Section IV.

A. Description of the Algorithm

RaQuN takes as input a set $\mathcal{M} = \{M_1, ..., M_n\}$ of n input models and returns a set T of matches (i.e., a matching). The

algorithm is divided into three phases. The goal of the first two phases (candidate initialization and candidate search) is to reduce the number of comparisons required in the third phase (matching).

Algorithm 1 RaQuN

```
1:  procedure RaQuN(M)                                  ▷ A set of input models
2:      E ← ⋃_{i=1}^{i=N} M_i                           ▷ Phase 1: Candidate Initialization
3:      tree ← createEmptyTree()
4:      for e ∈ E do
5:          v_e ← vectorize(e)
6:          tree ← insert(tree, e, v_e)
7:      end for
8:      P ← ∅                                           ▷ Phase 2: Candidate Search
9:      for e ∈ E do
10:         Nbrs ← neighborSearch(tree, e)
11:         for nbr ∈ Nbrs do
12:             p ← {e, nbr}
13:             if isValid(p) then
14:                 P ← P ∪ {p}
15:             end if
16:         end for
17:     end for
18:     P̂ ← filterAndSort(P)                            ▷ Phase 3: Matching
19:     T ← {{e} | e ∈ E}
20:     for {e, e'} ∈ P̂ do
21:         t ← select t ∈ T for which e ∈ t
22:         t' ← select t' ∈ T for which e' ∈ t'
23:         t̂ ← t ∪ t'
24:         if isValid(t̂) and shouldMatch(t, t', e, e') then
25:             T ← (T \ {t, t'}) ∪ {t̂}
26:         end if
27:     end for
28:     return T                                        ▷ The calculated matching
29: end procedure
```

Candidate Initialization (Line 2–7): In the first phase, RaQuN constructs a multi-dimensional search tree comprising all the elements of all input models as numerical vector representations. First, RaQuN collects the elements of all input models in an element set E, and initializes an empty tree. For each element $e \in E$, a vector representation v_e is determined and inserted into the tree. Hereby, each element is mapped to a specific point in the tree's vector space.

Candidate Search (Line 8–17): In the second phase, RaQuN determines promising match candidates by considering elements that are close to each other in the vector space, regarding a suitable distance metric (e.g., Euclidean distance). More specifically, RaQuN retrieves the k' nearest neighbors Nbrs for each element $e \in E$ in the vector space through a k'-NN search on the tree [41]. For every neighbor nbr $\in$ Nbrs of e, RaQuN creates an unordered pair $p = \{e, \text{nbr}\}$. If p is a valid match according to (1) (i.e., the two elements belong to different models), p is added to the match candidates P.

Matching (Line 18–27): In the final phase, RaQuN matches elements to each other by comparing the elements in the pairs P directly. First, in Line 18, all candidate pairs in P are sorted descendingly by their similarity, yielding list $\hat{P}$, omitting pairs with no common properties. Next, RaQuN creates a set T of matches such that each element $e \in E$ appears in exactly one single-element match $\{e\}$. The set T is a valid matching in which none of the elements has a corresponding partner. For every candidate match $\{e, e'\} \in \hat{P}$, RaQuN selects the two matches t and t' from T which contain the two elements e and e', respectively. Since every element $e \in E$ is in exactly one match in T, the selection of t and t' is unique. If the union $\hat{t}$ of t and t' is a valid match and its elements form a good match according to *shouldMatch*, RaQuN updates the matching T by replacing the two selected matches t and t' with the merged match $\hat{t}$. The algorithm terminates once all pairs in $\hat{P}$ have been processed. Each match now contains between one and n elements, and T represents a valid matching.

B. Exemplary Illustration

We illustrate RaQuN by applying it to our running example shown in Fig. 1, comprising the input models: $\mathcal{M} = \{\{1,2,3\},\{4,5,6\},\{7,8\}\}$.

Candidate Initialization: RaQuN first creates the set of all elements $E = \{1,2,3,4,5,6,7,8\}$ by forming the union over the models in $\mathcal{M}$. For our example, we choose a very simple two-dimensional vectorization. The first dimension is the average length of an elements' property names, and the second one is the number of properties of an element. Class *1:History-A*, for example, has an average property name length of 9.2 and five properties in total; its vector representation is $(9.2,\ 5)$. Fig. 2 visualizes the resulting k-dimensional vector space (k=2) and the points of all elements in E. We can see that intuitively corresponding classes are mapped to points close to each other, such as the two 'Physician' classes from model A and B.

Candidate Search: RaQuN performs range queries on the tree to find possible match candidates. For our example, we assume that the candidate search is configured to search for the three nearest neighbors of each element (k'=3). It is possible that multiple elements have the same vector representation and are mapped to the same point in the vector space, such as elements 3 and 5 in Fig. 2. Therefore, RaQuN might retrieve more than k' neighboring elements. In our example, RaQuN finds the neighbors $\{2,4,1\}$ for element *2:Physician-A*, and the neighbors $\{3,5,7,1\}$ for element *3:AdminAssistant-A*. Neighbors forming a valid match with the initial element can be considered as match candidates. For *3:AdminAssistant-A*, the retrieved candidate pairs are $\{3,5\}$ and $\{3,7\}$. Once the candidate search has been completed for all elements, we obtain the set P of candidate pairs:

$$P = \big\{\{1,4\},\{2,4\},\{3,5\},\{3,7\},\{5,7\},\{5,1\},\{6,2\},\{6,8\},\{7,1\},\{8,2\},\{8,4\}\big\}.$$

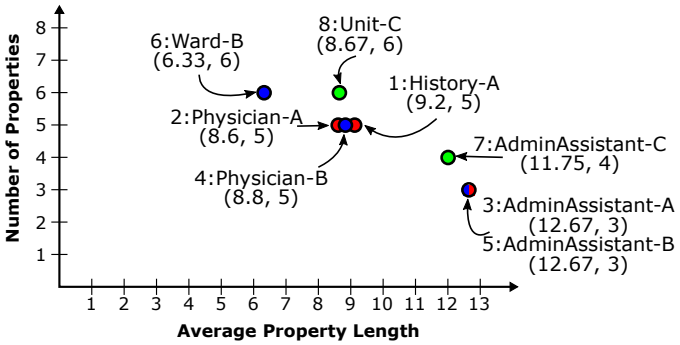

Fig. 2. Model elements of our running example mapped to points in a k-dimensional vector space (with k=2).

Matching: RaQuN sorts the match candidates P by descending confidence that their elements should be matched, according to its similarity function. For the sake of illustration, we choose a naive similarity function: the ratio of shared properties to all properties in the two elements. We receive the following (partially) sorted list of candidate pairs:

$$\hat{P} = \big(\{3,7\}{:}\tfrac{3}{4}, \{2,4\}{:}\tfrac{4}{6}, \{3,5\}{:}\tfrac{2}{4}, \\ \{5,7\}{:}\tfrac{2}{5}, \{7,1\}{:}\tfrac{1}{8}, \{6,8\}{:}\tfrac{1}{11}\big),$$

where $\{x,y\}{:}z$ denotes a pair with elements x and y having a similarity of z. Pairs with a similarity of 0 are removed during sorting, as their elements have no common properties.

Next, RaQuN initializes the set of matches T such that there is exactly one initial match for each element: $T = \big\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{7\},\{8\}\big\}$. RaQuN now iterates over the pairs in $\hat{P}$ and merges the corresponding matches in T accordingly. To keep the example simple, we assume that matches should be merged if the similarity of the candidate pair is at least $\frac{1}{2}$. The first pair that is selected is $\{3,7\}$, as its elements have the highest similarity. Thus, we select the matches $t = \{3\}$ and $t' = \{7\}$ from T and check whether their comprised elements should be matched. This is the case for the selected matches since the similarity between its elements is $\frac{3}{4} > \frac{1}{2}$. We thus merge the matches to the new match $\hat{t} = \{3,7\}$. We replace t and t' with $\hat{t}$, and receive $T = \big\{\{1\},\{2\},\{3,7\},\{4\},\{5\},\{6\},\{8\}\big\}$. In the second iteration, RaQuN selects $t = \{2\}$ and $t' = \{4\}$. Both are merged to the valid match $\hat{t} = \{2,4\}$. RaQuN repeats this process until all candidate matches in $\hat{P}$ have been considered. We obtain the final matching $T = \big\{\{1\},\{2,4\},\{3,5,7\},\{6\},\{8\}\big\}$, which is equal to the intuitive matching illustrated in Fig. 1.

C. Worst-Case Complexity

We estimate RaQuN's worst-case runtime complexity for each phase. Let n denote the number of input models and m the number of elements in the largest model.

Candidate Initialization: Each element $e \in E$ with $|E| \leq nm$ is vectorized and inserted into the tree. We assume that vectorization is an $O(1)$ operation. Given that insertion into a search tree is possible in $O(nm)$ [41], the worst-case runtime complexity of this phase is $O(nm \cdot (1 + nm)) = O(n^2m^2)$.

Candidate Search: For each of the at most nm elements in E, a neighbor search is performed which is possible in $O(\log nm)$ [41], [42]. For each of the potential nm neighbors (e.g., when all elements are at the same point) three constant runtime operations are performed in Line 12–15. This results in a complexity of $O(nm \cdot (\log nm + nm \cdot 1)) = O(n^2m^2)$.

Matching: The matching phase operates on the set of possible pairs $\hat{P}$ to match. In worst case, all elements from other models are valid match candidates for an element e during Phase 2. Thus, $|\hat{P}| \leq (nm)^2$ and sorting $\hat{P}$ in Line 18 requires $O(n^2m^2 \log nm)$ steps in the worst case. Constructing T in Line 19 is possible in $O(nm)$. The steps inside the loop at Line 20 have to be repeated $O(n^2m^2)$ times because $|\hat{P}| \leq (nm)^2$. Searching for matches t, t' in Line 21 and 22 has a worst case complexity of $O(nm)$ because $|T| \leq nm$. Merging the matches in Line 23 is $O(n)$ as valid matches only contain at most one element per model, i.e., $|t|, |t'| \leq n$. For the same reason, *shouldMatch* in Line 24 requires $O(n)$ steps. Line 25 exhibits worst-case runtime of $O(nm)$. We get $O(n^2m^2 \log nm + n^2m^2 \cdot nm) = O(n^3m^3)$.

Overall Complexity: The matching phase dominates the runtime complexity: We get $O(n^2m^2 + n^2m^2 + n^3m^3) = O(n^3m^3)$ in the worst case, which is an improvement over NwM's worst-case complexity of $\mathcal{O}(n^4m^4)$ [25]. In practice, we expect a much lower runtime complexity because Phase 1 and 2 of RaQuN are dedicated to reduce the number of comparisons in Phase 3. It is highly unlikely that all elements are mapped to the same point in the vector space such that all pairs of elements become potential match candidates in $\hat{P}$.

IV. Configuration Options

In this section, we discuss the variation points of RaQuN. For each of them, we propose a domain-agnostic configuration option such that RaQuN can be applied to models of any type.

A. Candidate Initialization

The candidate initialization has two points of variation: the *multi-dimensional search tree* and the *vectorization*.

RaQuN can construct the vector space with any multi-dimensional data structure supporting *insertion* and *neighbor search*, such as kd-trees [41].

The vectorization function defines the abstraction of model elements and their properties. Generally speaking, a vectorization function should cluster similar elements in the same region of the vector space. Clustering of similar elements becomes more likely with a higher number of dimensions, as more information about an element can be preserved in its vector representation. However, a higher number of dimensions also has a negative impact on the performance of the nearest neighbor search.

While we leave an in-depth investigation of this trade-off for future work, in this paper, we use a vectorization which can be applied to any element/property representation of a model. It represents all distinct properties of model elements of all input models by a dedicated dimension of the vector space $\{0,1\}^K$, where K is the number of distinct properties in all elements.

An element is represented by a bit vector in this space; the value at the index representing a dedicated property is set to 1 if the element has that property, and 0 otherwise.

B. Candidate Search

The candidate search is configured by the *number of considered nearest neighbors* k' and the *distance metric*.

The parameter k' determines how many neighbors are retrieved for each element, which directly influences how many candidate pairs p are considered during the matching phase. Increasing k' leads to more candidate pairs. Each neighbor will be less significant than the previous one as nearer (more similar) neighbors are considered first. While an optimal value of k' can only be determined empirically with respect to a dedicated measure of matching quality, a reasonable starting point for this is to set k'=n, as in our illustration in Section III-B. The rationale behind this is that each element may have at most one corresponding element per input model, limiting the number of corresponding elements to n-1. The choice of n respects that the nearest neighbor search considers the query point itself as first neighbor.

The distance metric is used to determine the distance between the vector representations of two elements in the vector space. The metric influences which elements are considered close or distant to each other (i.e., which elements are considered to be neighbors). In this work, we use the Euclidean distance, leaving experimentation with other distance metrics such as Cosine similarity or any custom metric (e.g., a metric emphasizing specific dimensions) for future work.

C. Candidate Matching

In RaQuN's final matching phase, potential match candidates are compared directly according to their *similarity*, and the *shouldMatch* predicate determines whether candidates should be formed to actual matches.

The *similarity* function is applied to assess the quality of a matching as illustrated in Section II. It is used to sort the match candidates $\hat{P}$ in Line 18 such that more similar pairs are considered to be merged first. One such similarity function is the *weight* metric by Rubin and Chechik [25], which assigns a weight $w(t) \in [0, 1]$ to a match depending on the number of common properties and the number of elements in the match. Given a match t, the weight is calculated as

$$w(t) = \frac{\sum_{2 \leq j \leq |t|} j^2 \cdot n_j^p}{n^2 \cdot |\pi(t)|} \tag{2}$$

where $|t|$ denotes the size of the match, n_j^p the number of properties that occur in exactly j elements of the match, and $\pi(t)$ is the set of all distinct properties of all elements in t.

The purpose of *shouldMatch* is to compensate potential inaccuracy from abstracting elements by numerical vectors. In general, an implementation of *shouldMatch* could work on concrete model representations, consider meta-data related to the models, etc. To stay independent of such domain-specific aspects, in this work, we stick to using the generic weight metric for implementing *shouldMatch*. The idea is that any extension of a match should increase the quality of the overall matching. To that end, two matches t and t' are merged if the weight of the merged match $t \cup t'$ is greater than the sum of the individual match weights:

$$shouldMatch(t, t', e, e') := w(t \cup t') > w(t) + w(t') \tag{3}$$

TABLE I
EXPERIMENTAL SUBJECTS AND THEIR CHARACTERISTICS.

	Model Type	#Models	Elements Avg.	Elements Median	Properties Avg.	Properties Median
Hospital	Simple class diag.	8	27.62	26	4.84	4
Warehouse	Simple class diag.	16	24.25	22	3.65	3
Random	Synthetic	100	26.99	26	5.36	5
Loose	Synthetic	100	28.88	29	4.43	4
Tight	Synthetic	100	25.01	25	8.79	9
PPU Structure	SysML block diag.	13	32.15	32	3.26	2
PPU Behavior	UML statemachines	13	221.85	228	5.04	5
bCMS	UML class diag.	14	67.71	63	3.60	2
BCS	Component/connector	18	78.78	72	5.81	4
ArgoUML	UML class diag.	7	1,752.86	1,749	9.05	4
Apo-Games	Simple class diag.	20	63.05	60	19.62	13

V. EVALUATION

In addition to our conceptual and theoretical contributions, we conduct an empirical investigation on a variety of datasets. We are interested in whether RaQuN scales for large models while achieving high matching quality. The full replication package can be found on Zenodo [43].

RQ1 Does RaQuN achieve better runtime and/or matching quality compared to the simultaneous n-way matching algorithm NwM and sequential two-way approaches?

RQ2 How does the configuration of RaQuN's candidate search affect its performance?

RQ3 How does RaQuN scale with growing model sizes?

A. Selected Algorithms

We compare RaQuN with NwM and two sequential two-way approaches (Pairwise). All algorithms are implemented in Java and use the weight metric [25] (2) as a similarity function (see Section II).

1) Prototypical Implementation of RaQuN: We implemented a prototype of RaQuN which uses a generic kd-tree implemented by the Savarese Software Research Corporation [44] as multi-dimensional search tree. For all other variation points, we implemented the domain-agnostic configuration options discussed in Section IV.

2) Baseline Algorithms: The prevalent way to calculate n-way matchings is sequential two-way [19]–[24]. This leaves open (a) which two-way matching algorithm is used in each iteration, and (b) the order in which inputs are processed. For (a) we use the Hungarian algorithm [45] to maximize the weight of the matching in each iteration. For (b), Rubin and Chechik [25] report the most promising results for the *Ascending* and *Descending* strategies, which sort the input models by number of elements in ascending and descending order, respectively. For NwM, we use the prototype implementation provided by Rubin and Chechik [25].

B. Experimental Subjects

Our experimental subjects and their basic characteristics are summarized in Table I. Converting the experimental subjects into element/property representations requires a pre-processing step that is model-type and technology-specific. We used the generic EMF model traversal and reflective API to access an element's local properties and referenced elements. Our pre-processing code is part of our replication package [43].

1) Experimental Subjects of Rubin and Chechik: To enable a fair comparison with NwM, the first five subjects selected for our evaluation stem from the n-way model matching benchmark set used by Rubin and Chechik [25]. The *Hospital* and *Warehouse* datasets include sets of student-built requirements models of a medical information and a digital warehouse management system, for both of which variation arises from taking different viewpoints. Both datasets originate from case studies conducted in a Master's thesis by Rad and Jabbari [34]. The latter three datasets have been synthetically created using a model generator, which in the *Random* case mimics the characteristics of the hospital and warehouse models. The *Loose* scenario exposes a larger range of model sizes and a smaller number of properties shared among the models' elements, while the *Tight* scenario exposes a smaller range w.r.t. these parameters.

2) Variant Sets Generated from Software Product Lines: The second set of selected subjects are variant sets generated from model-based software product lines. We use a superset of the n-way model merging benchmark set used in a recent work of Reuling et al. [46].

The Pick and Place Unit (*PPU*) is a laboratory plant from the domain of industrial automation systems [47], [48] whose system structure and behavior are described in terms of SysML block diagrams and UML statemachines, respectively [35]. Variation arises from different scenarios supported by the plant. The Barbados Car Crash Crisis Management System (*bCMS*) [36], [49] supports the distributed crisis management by police and fire personnel for accidents on public roadways. We focus on the object-oriented implementation models of the system [36], including both functional and non-functional variability. The Body Comfort System (*BCS*) [37] is a case study from the automotive domain whose software can be configured w.r.t. the physical setup of electronic control units. We use the component/connector models of BCS, specifying the software architecture of the 18 variants sampled by Lity et al. [37]. *ArgoUML* is a publicly available CASE-tool supporting model-driven engineering with the UML. It was used in prior studies [50], [51] and provides a ground truth for assessing the quality of a matching using precision and recall. The dataset comprises detailed class models of the Java implementation [38]. They represent different tool variants which have been extracted by removing specific features for supporting different UML diagrams.

3) Variant Sets Created Through Clone-and-Own: The last experimental subject stems from a software family called *Apo-Games* which has been developed using the clone-and-own approach [1], [39] (i.e., new variants were created by copying and adapting an existing one) and which has been recently presented as a challenge for variability mining [52]. The challenge comprises 20 Java and five Android variants, from which we selected the Java variants only.

C. Evaluation Metrics

While measuring efficiency is a largely straightforward micro benchmarking task, there exists no generally accepted definition of the quality of a matching in the literature [46]. We use the two most widely established quality evaluation metrics *weight* and *precision/recall*.

1) Weight: One way to measure the quality of an n-way matching is the weight metric [25], where the optimal matching is the one with the highest weight, expressed as the sum of the individual match weights. Given a matching T, its weight is calculated as $w(T) = \sum_{t \in T} w(t)$, where $w(t)$ is calculated as in (2). There can be several matchings with the same weight, and thus several optimal matchings for a set of models. We selected the weight metric as it does not depend on a ground truth, which is often not available.

2) Precision/Recall: In the context of two-way matching, the quality of a matching is often assessed using oracles and traditional measures (i.e., precision and recall) known from the field of information retrieval [53]. For our experimental subjects, however, such oracles are only available for models generated from an SPL. Here, unique identifiers $id(e)$ may be attached to all model elements e of the integrated code base and serve as oracles when being preserved by the model generation. This way, corresponding elements have the same ID. These IDs are generally not available for models that did not originate from an SPL (e.g., models created through cloning), and they are not exploited by the matching algorithms used in our experiments.

Each two-element subset of a valid match is considered a true positive TP if its elements share the same ID. If these elements have different IDs, they are considered false positive FP. Two elements sharing the same ID but being in distinct matches are considered false negatives FN. The amount of TP, FP, and FN is defined over all the matches in T:

$$TP(T) = \sum_{t \in T} \left| \left\{ \{e_1, e_2\} \subseteq t \mid id(e_1) = id(e_2) \right\} \right| \quad (4)$$

$$FP(T) = \sum_{t \in T} \left| \left\{ \{e_1, e_2\} \subseteq t \mid id(e_1) \neq id(e_2) \right\} \right| \quad (5)$$

$$FN(T) = \Big| \bigcup_{\substack{t_1, t_2 \in T \\ t_1 \neq t_2}} \left\{ \{e_1, e_2\} \mid e_1 \in t_1, e_2 \in t_2, id(e_1) = id(e_2) \right\} \Big| \quad (6)$$

Precision and recall are calculated as usual [53].

D. Methodology and Results

We run our experiments on a workstation with an Intel Xeon E7-4880 processor with a frequency of 2.90GHz. In order to reduce the influence of side-effects caused by additional workload on the experimental workstation, we run each algorithm 30 times on each of our experimental subjects, except for Random, Loose, and Tight for which we follow the

TABLE II
COMPARISON OF ACHIEVED WEIGHTS AND RUNTIMES ACROSS ALL ALGORITHMS, AVERAGED OVER 30 RUNS FOR EACH SUBJECT.

Algorithm	Hospital		Warehouse		Random		Loose		Tight		Apo-Games	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN	**4.92**	**0.08** [0.07, 0.12]	**1.63**	**0.32** [0.27, 0.93]	**1.04**	**0.07** [0.05, 0.13]	**1.03**	**0.13** [0.07, 0.33]	**0.94**	**0.07** [0.05, 0.11]	**18.27**	71.12 [41.30, 104.58]
NwM	4.49	20.64 [16.18, 24.22]	1.46	74.51 [29.53, 84.87]	0.80	24.00 [1.23, 76.02]	0.79	22.40 [1.12, 70.68]	0.88	39.75 [1.63, 66.60]	17.91	5,462.91 [3,742.22, 6,597.83]
Pairwise Ascending	4.49	0.31 [0.28, 0.44]	1.11	0.36 [0.33, 0.45]	0.79	0.21 [0.10, 0.31]	0.74	0.14 [0.08, 0.22]	**0.94**	0.22 [0.16, 0.34]	12.96	**10.42** [9.32, 12.04]
Pairwise Descending	4.72	0.16 [0.14, 0.22]	1.27	0.36 [0.32, 0.53]	0.78	0.15 [0.10, 0.23]	0.74	0.14 [0.08, 0.25]	0.93	0.22 [0.17, 0.33]	16.40	10.68 [9.27, 13.50]

Algorithm	PPU Structure		PPU Behavior		bCMS		BCS		ArgoUML	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN	**28.95**	0.85 [0.81, 0.90]	**164.53**	18.32 [16.39, 20.34]	**41.53**	2.84 [1.40, 3.58]	**51.17**	12.58 [8.10, 18.22]	**1727.65**	2,647.76 [1,483.48, 3,324.70]
NwM	28.64	9.27 [7.84, 10.07]	146.35	4,616.30 [3,410.86, 5,919.65]	41.25	247.31 [227.52, 275.95]	43.20	330.25 [235.74, 388.45]	- - - timeout - - -	
Pairwise Ascending	28.65	0.27 [0.22, 0.38]	145.46	11.03 [10.32, 12.61]	39.76	1.16 [1.13, 1.21]	43.83	5.69 [3.85, 7.43]	1702.91	318.26 [297.43, 379.76]
Pairwise Descending	28.89	**0.26** [0.21, 0.42]	142.56	**10.84** [10.20, 13.01]	38.76	**1.04** [1.01, 1.07]	47.16	**4.84** [3.37, 6.73]	1710.25	**314.88** [302.28, 329.35]

methodology of Rubin and Chechik [25]. Here, we select 10 subsets comprising 10 of the 100 models for each run that is repeated 30 times, leading to 300 runs per algorithm and subject. Regardless of the experimental subject, we permutate the input models randomly for each experimental run. We set a time-out of 12 hours for the matching of a dataset, due to the large amount of experimental runs.

1) RQ1: Runtime and Weight Compared to Other Matching Algorithms: Table II presents the average weight and runtime achieved by the algorithms on each of our experimental subjects. Here, we consider the weight metric as it does not require a ground truth, which is not available for all datasets.

RaQuN and Pairwise are significantly faster than NwM. While, on average, NwM requires between 9s and 75s for the matching of smaller subjects (< 50 elements) PPU Structure and Hospital through Tight, the other algorithms are able to calculate a matching in less than a second. Matchings for bCMS and BCS were calculated by RaQuN and Pairwise in less than 13s, where NwM required 247s and 330s. Moreover, NwM was not able to provide a matching for ArgoUML before reaching the time-out of 12h, and it took about 90min and 70min for processing Apo-Games and PPU Behavior, respectively. In contrast, RaQuN provides a matching in an average time of less than 45min for ArgoUML, 71s for Apo-Games, and less than 30s for PPU Behavior.

When considering the achieved weights, RaQuN delivers the matchings with the highest weights for all datasets. NwM delivers higher weights than Pairwise for six of the eleven datasets. Notably, ascending and descending Pairwise always yield different weights, which confirms the observation by Rubin and Chechik that performance of sequential two-way matchers depends on the order of input models [25].

RaQuN is significantly faster than NwM on all datasets, and is almost as fast as two-way matchers on medium-sized datasets with hundreds of elements. Moreover, RaQuN achieves the highest weights across all datasets.

2) RQ2: Configuration of the Candidate Search: In order to assess how the configuration of the candidate search influences RaQuN's performance, we ran RaQuN with increasing values of k' for the nearest neighbor search. Fig. 3 presents the results of the runs conducted on the datasets PPU, bCMS, and ArgoUML. The plots display the value of k' against the runtime of RaQuN. The red line marks the k' at which the candidate search retrieved all match candidates necessary to reach the peak weight performance of RaQuN. The blue line marks the k' that is equal to the number of models n, which we propose as a possible heuristic for k'.

Our findings show that setting $k' = n$ made it possible to achieve the best matching possible with RaQuN. RaQuN was able to find the best candidates with small values of k', due to multiple elements being mapped to the same neighboring point in the vector space, and due to considering the neighbors for each element individually. Selecting a higher value for k' does not deteriorate the match quality, because the final match decision depends on *shouldMatch*. Moreover, the runtime of RaQuN shows a linear growth with higher k', which indicates that considering more neighbors than necessary will not cause a sudden increase in runtime.

Table III presents an overview of how many comparisons are saved by the candidate search (using k'=n). For most experimental subjects, RaQuN is able to reduce the number of comparisons by more than 90%. PPU is the only subject on which we achieve a rather low reduction of 48.5%. This is due to the high similarity between the elements, and the fact that the models are relatively small.

With the heuristic choice of k'=n, RaQuN retrieves enough candidates for good matches, while still reducing the number of element comparisons by more than 90% for most experimental subjects.

TABLE III
NUMBER OF ELEMENT COMPARISONS THAT ARE SAVED BY RAQUN WITH k'=n. THE REDUCTION IS ACHIEVED THROUGH RAQUN'S NEAREST NEIGHBOR SEARCH ON A MULTIDIMENSIONAL TREE.

Dataset	Full N-Way Matching	RaQuN	
	#Comparisons	#Comparisons	Saved
Hospital	21,211	1,229	94.2%
Warehouse	70,037	6,408	90.9%
Random	33,600	2,499	92.6%
Loose	26,244	2,670	89.8%
Tight	25,237	2,215	91.2%
PPU Structure	80,620	41,533	48.5%
PPU Behavior	3,814,644	259,827	93.2%
bCMS	416,571	106,585	74.4%
BCS	939,346	253,862	73.0%
ArgoUML	64,521,622	481,179	99.3%
Apo-Games	750,319	31,880	95.8%

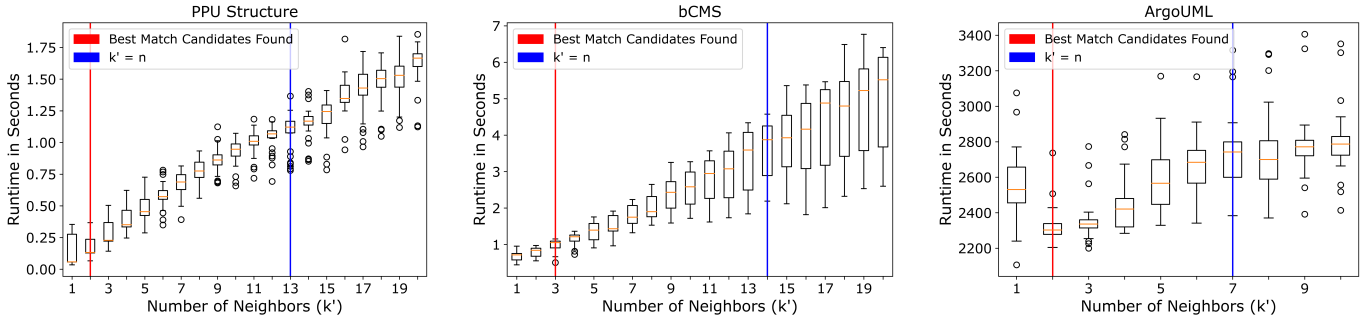

Fig. 3. Impact of an increasing number of neighbors considered for matching on the performance of RaQuN.

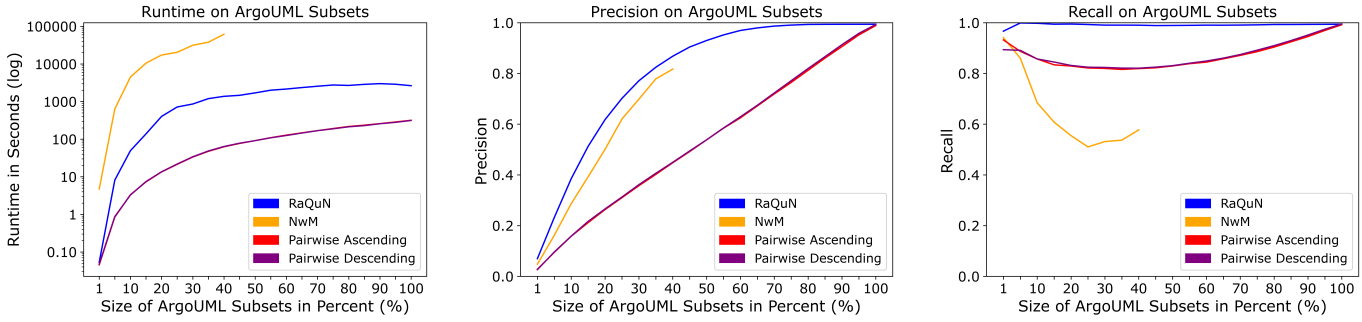

Fig. 4. Average precision, recall and runtime of RaQuN, NwM, and Pairwise on subsets of ArgoUML with increasing size.

3) RQ3: Scalability with Growing Size of Input Models: As already mentioned, realistic applications of n-way matching in practice typically have to deal with large models but only a few model variants. Thus, we are primarily interested in how the algorithms scale with growing model sizes for a fixed number of model variants. Answering this question requires experimental subjects with a stepwise size increase, and cannot be answered by only considering the results shown in Table II.

To that end, in addition to the experimental subjects introduced in Section V-B, we generated subsets of ArgoUML, which comprises the largest models of our subjects. All subsets have the same number of models as ArgoUML but vary in the number of elements. The number of elements in each subset is a fixed percentage between 5% and 100% of the number of elements in ArgoUML. We increased the percentages in 5% steps and generated 30 subsets for each percentage, in addition to 30 subsets with 1% of elements. The sub-models are generated as follows. First, we randomly select a subset of classes from the set of all classes of a given model such that the subset contains the desired percentage of the overall number of classes. We repeat this for each model in ArgoUML so that the number of models remains the same. Second, we eliminate properties corresponding to dangling references in the selected classes, such that no typed property references a class which is not contained in the subset of selected classes.

The results of our scalability analysis on these subsets are shown in Fig. 4. The leftmost plot presents the average logarithmic runtimes of the algorithms for each subset size. We observe that the runtime of NwM increases rapidly with the subset size. NwM requires more than 60 minutes on average to compute a matching on the 15% subsets. This confirms that it is not feasible to match larger models with NwM. In contrast, it is still feasible to run RaQuN and Pairwise on the full ArgoUML models. For matching the full models (cf. Table II), the average runtime of RaQuN was less than 45 minutes, making it slower than Pairwise, but still feasible.

Yet, faster runtimes are usually achieved at the cost of matching quality. Therefore, we also evaluate the quality of the computed matchings to confirm that RaQuN delivers matches of higher quality than Pairwise on all subsets. To asses the quality, we cannot use the weight metric because the weight of a match directly depends on the size of the match. Weights are thus not comparable across (sub-)models of increasing size. Instead, we assess precision and recall, which is possible for ArgoUML due to the availability of a ground truth.

The precision achieved by the different algorithms is presented in the central plot of Fig. 4. On the subset with only 1% of elements, the matching precision of all approaches lies at roughly 0.1. With increasing subset size, we note a significant difference in precision when we compare the n-way and sequential two-way approaches. Moreover, we can observe a slightly higher precision for RaQuN in comparison to NwM. The n-way algorithms deliver more precise matchings because Pairwise does not consider all possible match candidates for an element at once and therefore may form worse matches.

Lastly, the rightmost plot of Fig. 4 shows the recall achieved by the algorithms. For all algorithms, the recall first drops with increasing subset size and then rises again after reaching a sub-

set size between 30% and 50%, depending on the algorithm. The reason for this is that, according to our ArgoUML subset generation, the number of elements initially grows faster than the number of properties of each element. The latter depends on the occurrence of other types in the model which may not be included in the sub-model yet. As a consequence, some matches are missed. While this effect is only barely noticeable for RaQuN, it is prominent for NwM and Pairwise. The comparably high recall achieved by RaQuN indicates that the vectorization is able to mitigate this effect. On the other hand, Pairwise shows a larger drop in recall, as forming incorrect matches (see precision) can additionally impair its ability to find all correct matches. To our surprise, the recall of NwM drops significantly more than the recall of Pairwise. We assume that the optimization step of NwM, which may split already formed matches into smaller ones, accounts for a higher loss in recall.

As opposed to NwM, RaQuN shows good scaling properties for models of increasing size, up to the largest models of our experimental subjects containing more than 10,000 elements in total. This is comparable to Pairwise, and an indicator that RaQuN's typical scaling behavior is considerably better than its theoretical worst case complexity. At the same time, RaQuN is able to deliver matchings of higher quality (w.r.t. precision and recall).

E. Threats to Validity

Our experiments rely on evaluation metrics that may affect the *construct validity* of the results. First, we use the weight metric which has already been used in prior studies [25]. While it can be applied to compare the results on the same subject, weights obtained for different experimental subjects are hardly comparable. To that end, we use precision and recall [53] in order to asses the quality of the matchings for the ArgoUML subsets. The calculation of both depends on our definition of true positives, false positives, and false negatives. Here, we favored a pairwise comparison over a direct rating of complete matches to rate almost correct matches better than completely wrong matches. Another potential threat pertains the construction of ArgoUML subsets. Using unrelated models of different size would introduce the bias of varying characteristics of these models. Hence, we decided to remove parts of the largest available system. While we argue that this is the better choice, it is possible that the ArgoUML subsets do not represent realistic models. Moreover, using the product-line variants of ArgoUML, PPU, BCS, and bCMS as experimental subjects could have introduced a bias because these are derived from a clean and integrated code base, lacking unintentional divergence [54]–[56]. Thus, while it is common in the literature to use product-line datasets [50], [51], [57], [58] as they inherently provide ground-truth matchings, we also considered the clone-and-own system ApoGames.

Computational bias and random effects are a threat to the *internal validity*. Other processes on the machine may affect the runtime, but also the matching may differ in several runs with the same input. The non-determinism of RaQuN is due to the use of hash sets used in the implementation. Furthermore, the order in which matches are merged may vary for identical similarity scores. We mitigated those threats by repeating every measurement 30 times, each with a different permutation of the input models. Additionally, the random generation of the ArgoUML subsets might have introduced a bias favoring a particular algorithm. To mitigate this bias, we sampled 30 subsets for each subset size, totaling in 600 different subsets included in the replication package. Faults in the implementation may also affect the results. We implemented several unit tests for each class of RaQuN's implementation and manually tested the quality of RaQuN and the evaluation tools on smaller examples. Additionally, we resort to the original implementations of NwM and Pairwise.

The question whether the results generalize to other subjects, which first must be converted to element/property models, is a threat to the *external validity*. We mitigate this threat by our selection of diverse experimental subjects. We used the experimental subjects from the original evaluation of NwM, for which Rubin and Chechik have already mitigated this threat [25]. Moreover, we have experimented with additional subjects covering (a) *different domains*, i.e., information systems (bCMS), industrial plant automation (PPU), automotive software (BCS), software engineering tools (ArgoUML), and video games (Apo-Games), (b) *different origins*, i.e., academic case studies on model-based software product lines (PPU, BCS, bCMS), a software product line which has been reverse engineered from a set of real-world software variants written in Java (ArgoUML), and a set of variants developed using clone-and-own (Apo-Games), and (c) *different model types*, i.e., UML class diagrams (bCMS, ArgoUML), SysML block diagrams and UML statemachines (PPU), and component/connector models (BCS).

VI. Related Work

Traditional matchers are two-way matchers which can be classified into *signature-based*, *similarity-based*, and *distance-based* approaches. Signature-based approaches match elements which are "identical" concerning their signature [59], typically a hash value which comprises conceptual properties (e.g., names) or surrogates (e.g., persistent identifiers). Similarity-based matching algorithms try to match the most similar but not necessarily equal model elements [7]–[11], [17]. Distance-based approaches try to establish a matching which yields a minimal edit distance [12]–[16], [18]. Among these categories, signature-based matching is the only one which could be easily generalized to the n-way case. However, the limitations of signatures have been extensively discussed [8], [9], [31]–[33].

A few approaches realize n-way matching by the *repeated two-way matching* of the input artifacts [19]–[24]. However, as reported by Rubin and Chechik [25] and now confirmed by our empirical evaluation, this may yield sub-optimal or even incorrect results as not all input artifacts are considered at the same time [21], [25].

To the best of our knowledge, Rubin and Chechik are the only ones who have studied the *simultaneous* matching of n input models [25]. Their algorithm called NwM applies iterative bipartite graph matching whose insufficient scalability motivated our research. RaQuN is radically different from NwM. It is the first algorithm applying index structures (i.e., multi-dimensional search trees) to simultaneous n-way model matching (Phase 1 and Phase 2 in Alg. 1). Even without these phases, the matching (Phase 3) differs from NwM by abstaining from bipartite graph-matching, reducing the worst-case complexity (see Section III-C).

Our usage of multi-dimensional search trees is inspired by Treude et al. [31]. While they discuss basic ideas of how model elements can be mapped onto numerical vectors in the context of two-way matching, the actual matching problem was not even addressed but delegated to an existing two-way matcher. Moreover, a dedicated vectorization function needs to be provided for all types of model elements, while we work with a vectorization which is domain-agnostic.

All approaches to both n-way and two-way matching assume matches to be mutually disjoint and that no two elements of a match belong to the same input model. This is a reasonable assumption which we adopt in this paper to ensure the comparability of RaQuN with the state-of-the-art. The only exception which deviates from this assumption is the distance-based two-way approach presented by Kpodjedo et al. [60], which extends an approximate graph matching algorithm to handle many-to-many correspondences. Regarding the ground truth matchings of our experimental subjects obtained from product lines, there is no need for such an extension of n-way matching algorithms. However, it might be a valuable extension for some use cases (e.g., for comparing models at different levels of abstraction) which we leave for future work.

Several approaches which can be characterized as *merge refactoring* have been proposed in the context of migrating a set of variants into an integrated software product line. Starting from a set of "anchor points" which indicate corresponding elements, the key idea is to extract the common parts in a step-wise manner through a series of variant-preserving refactorings [26], [39], [46], [58], [61]–[66]. Anchor points may be determined through clone detection [39], [62]–[64] or conventional matchers [26], [46], [61], [65], [66], and may be corrected and improved by the merge refactoring. However, such implicit calculations of optimized n-way matchings require extensive catalogues of language-specific refactoring operations which have to be specified manually [46], [63]–[65]. Merge refactoring approaches are complementary to our approach, because they require sufficiently accurate matchings as input to avoid prohibitive computational efforts during refactoring [46].

Another approach for managing cloned software variants has been presented by Linsbauer et al. [30], [51]. They use combinatorics of feature configurations to map features to parts of development artifacts, which implicitly establishes n-way matchings. Similarly, implicit n-way matchings are established through extracting product-line architectures as, e.g., proposed by Assunção et al. [27]. However, the required additional information such as complete feature configurations is typically not available.

Finally, Babur et al. [28], [29] cluster models in model repositories for the sake of repository analytics. They translate models into a vector representation to reuse clustering distance measures. However, clustering is performed on the granularity level of entire models, while our candidate initialization clusters individual model elements. In fact, as shown by Wille et al. [67], both may be used complementary by first partitioning a set of model variants and then performing a fine-grained n-way matching on clusters of similar models.

VII. Conclusion and Future Work

Model matching is a major requirement in many fields, including extractive software product-line engineering and multi-view integration. In this paper, we proposed RaQuN, a generic algorithm for simultaneous n-way model matching which scales for large models. We achieved this by indexing model elements in a multi-dimensional search tree which allows for efficient range queries to find the most suitable matching candidates. We are the first to provide a thorough investigation of n-way model matching on large-scale subjects (ArgoUML) and a real-world clone-and-own subject (ApoGames). Compared to the state-of-the-art, RaQuN is an order of magnitude faster while producing matchings of better quality. RaQuN makes it possible to adopt simultaneous n-way matching in practical model-driven development, where models serve as primary development artifacts and may easily comprise hundreds or even thousands of elements.

Our roadmap for future work is threefold. First, regarding the various variations points of the generic matching algorithm, we plan an in-depth investigation of RaQuN's configurability w.r.t. potential domain-specific optimizations. For example, RaQuN could be adjusted to specific requirements of different application scenarios and characteristics of different types of models. Second, RaQuN, Pairwise, and NwM only support matching one element of a model to at most one element of each other model (1-to-1). This might limit the possibility to find the correct matches in certain cases (e.g., an element was split into several smaller elements). Therefore, from a more general point of view, we want to extend simultaneous n-way model matching to support n-to-m matches for which we believe that RaQuN serves as a promising basis to enter and explore this new aspect of n-way matching. Third, in accordance with the state-of-the-art, RaQuN forms mutually disjoint matches. Therefore, an element belongs to at most one match and no alternative matches for an element are computed. We plan on supporting scenarios in which several match proposals instead of a single exact match for a specific element are desired (e.g., scenarios in which a user interactively selects the most suitable match for an element).

Acknowledgment

We used Matplotlib [68] to create the plots that are shown in Fig. 3 and Fig. 4.

References

[1] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[2] S. Feldmann, J. Fuchs, B. Vogel-Heuser *et al.*, "Modularity, variant and version management in plant automation–future challenges and state of the art," in *Proceedings of the 12th International Design Conference, Dubrovnik, Croatia*, 2012, pp. 1689–1698.

[3] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[4] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A Survey of Variability Modeling in Industrial Practice," in *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2013, pp. 7:1–7:8.

[5] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Software Engineering (TSE)*, vol. 28, no. 5, pp. 449–462, May 2002.

[6] M. Sabetzadeh and S. Easterbrook, "View Merging in the Presence of Incompleteness and Inconsistency," *Requirements Engineering*, vol. 11, no. 3, pp. 174–193, 2006.

[7] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2005, pp. 54—65.

[8] U. Kelter, J. Wehren, and J. Niere, "A Generic Difference Algorithm for UML Models." *Software Engineering*, pp. 105–116, 2005.

[9] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching," in *Proc. Int'l Conf. on Data Engineering (ICDE)*. IEEE, 2002, pp. 117–128.

[10] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave, "Matching and Merging of Statecharts Specifications," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2007, pp. 54—-64.

[11] C. Brun and A. Pierantonio, "Model Differences in the Eclipse Modeling Framework," *The European Journal for the Informatics Professional*, vol. 9, no. 2, 2008.

[12] W. Miller and E. W. Myers, "A File Comparison Program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

[13] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An Enhanced Line Differencing Tool," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2009, pp. 595–598.

[14] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, "LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 2013, pp. 230–239.

[15] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Software Engineering (TSE)*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[16] M. Kim, D. Notkin, and D. Grossman, "Automatic Inference of Structural Changes for Matching Across Program Versions," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2007, pp. 333–343.

[17] T. Apiwattanapong, A. Orso, and M. J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2004, pp. 2–13.

[18] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-Grained and Accurate Source Code Differencing," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, 2014, pp. 313–324.

[19] D. Wille, S. Schulze, C. Seidl, and I. Schaefer, "Custom-Tailored Variability Mining for Block-Based Languages," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2016, pp. 271–282.

[20] D. Wille, S. Schulze, and I. Schaefer, "Variability Mining of State Charts," in *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2016, pp. 63–73.

[21] S. Duszynski, "Analyzing Similarity of Cloned Software Variants Using Hierarchical Set Models," Ph.D. dissertation, University of Kaiserslautern, 2015.

[22] B. Klatt and M. Küster, "Improving Product Copy Consolidation by Architecture-Aware Difference Analysis," in *Proc. Int'l Conf. on Quality of Software Architectures (QoSA)*. ACM, 2013, pp. 117–122.

[23] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Automatic Library Migration for the Generation of Hardware-In-The-Loop Models," *Science of Computer Programming (SCP)*, vol. 77, no. 2, pp. 83–95, 2012.

[24] A. Schlie, S. Schulze, and I. Schaefer, "Recovering Variability Information from Source Code of Clone-and-Own Software Systems," in *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 2020, pp. 1–9.

[25] J. Rubin and M. Chechik, "N-Way Model Merging," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 301–311.

[26] D. Reuling, U. Kelter, S. Ruland, and M. Lochau, "SiMPOSE-Configurable N-Way Program Merging Strategies for Superimposition-Based Analysis of Variant-Rich Software," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1134–1137.

[27] W. K. G. Assunção, S. R. Vergilio, and R. E. Lopez-Herrejon, "Automatic Extraction of Product Line Architecture and Feature Models from UML Class Diagram Variants," *J. Information and Software Technology (IST)*, vol. 117, p. 106198, 2020.

[28] Ö. Babur, "Statistical Analysis of Large Sets of Models," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2016, pp. 888–891.

[29] Ö. Babur and L. Cleophas, "Using N-Grams for the Automated Clustering of Structural Models," in *Proc. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Springer, 2017, pp. 510–524.

[30] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "The ECCO Tool: Extraction and Composition for Clone-and-Own," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2015, pp. 665–668.

[31] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference Computation of Large Models," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2007, pp. 295–304.

[32] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige, "Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing," in *Proc. of the Workshop on Comparison and Versioning of Software Models (CVSM)*. IEEE, 2009, pp. 1–6.

[33] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt, "Adaptability of Model Comparison Tools," in *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 2012, pp. 306–309.

[34] 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, 2012.

[35] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann, "Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit," Institute of Automation and Information Systems, TU München, Tech. Rep., 2014.

[36] A. Capozucca, B. Cheng, N. Guelfi, and P. Istoan, "OO-SPL Modelling of the Focused Case Study," in *Proc. Int'l Workshop on Comparing Modeling Approaches (CMA)*. ACM, 2011.

[37] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer, "Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study," Technische Universität Braunschweig, Tech. Rep., 2012.

[38] J. Martinez, W. K. G. Assunção, and T. Ziadi, "ESPLA: A Catalog of Extractive SPL Adoption Case Studies," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2017, pp. 38–41.

[39] J. Rubin, K. Czarnecki, and M. Chechik, "Managing Cloned Variants: A Framework and Experience," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2013, pp. 101–110.

[40] S. Wenzel, "Unique identification of elements in evolving software models," *Software and System Modeling (SoSyM)*, vol. 13, no. 2, pp. 679–711, 2014.

[41] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[42] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 209—226, 1977.

[43] A. Schultheiß, P. M. Bittner, L. Grunske, T. Thüm, and T. Kehrer, "Artifact: Scalable N-Way Model Matching Using Multi-Dimensional Search Trees," Jul. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5150388

[44] S. Daniel F. Savarese, "libssrckdtree-j," Source Code, available online at https://www.savarese.com/software/libssrckdtree-j/; visited on December 28, 2020.

[45] H. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Nordic J. Computing*, vol. 2, no. 1-2, pp. 83–97, 1955.

[46] D. Reuling, M. Lochau, and U. Kelter, "From imprecise n-way model matching to precise n-way model merging," *J. Object Technology (JOT)*, vol. 18, no. 2, 2019.

[47] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of Software in Automated Production Systems: Challenges and Research Directions," *J. Systems and Software (JSS)*, vol. 110, pp. 54–84, 2015.

[48] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr, "Reasoning about Product-Line Evolution Using Complex Feature Model Differences," *Automated Software Engineering*, vol. 23, no. 4, pp. 687–733, 2015.

[49] A. Capozucca, B. Cheng, G. Georg, N. Guelfi, P. Istoan, G. Mussbacher, A. Jensen, J.-M. Jézéquel, J. Kienzle, J. Klein *et al.*, "Requirements Definition Document for a Software Product Line of Car Crash Management Systems," The Repository of Model-Driven Development (ReMoDD), 2011.

[50] C. Kästner, A. Dreiling, and K. Ostermann, "Variability Mining: Consistent Semiautomatic Detection of Product-Line Features," *IEEE Trans. Software Engineering (TSE)*, vol. 40, no. 1, pp. 67–82, 2014.

[51] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability Extraction and Modeling for Product Variants," *Software and System Modeling (SoSyM)*, vol. 16, no. 4, pp. 1179–1199, Oct. 2017.

[52] J. Krüger, W. Fenske, T. Thüm, D. Aporius, G. Saake, and T. Leich, "Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, Sep. 2018, pp. 251–256.

[53] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. New York, NY, USA: ACM, 1999, vol. 463.

[54] B. Klatt, M. Küster, and K. Krogmann, "A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies," in *Proc. Int'l Workshop on Reverse Variability Engineering (REVE)*, 2013, pp. 1–8.

[55] T. Schmorleiz and R. Lämmel, "Similarity Management via History Annotation," in *Proc. Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, 2014, pp. 45–48.

[56] T. Schmorleiz, "An Annotation-Centric Approach to Similarity Management," Master's thesis, Universität Koblenz-Landau, Germany, Feb. 2015.

[57] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach," in *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 2015, pp. 101–110.

[58] T. Ziadi, C. Henard, M. Papadakis, M. Ziane, and Y. Le Traon, "Towards a Language-Independent Approach for Reverse-Engineering of Software Product Lines," in *Proc. ACM Symposium on Applied Computing (SAC)*. ACM, 2014, pp. 1064–1071.

[59] P. Selonen and M. Kettunen, "Metamodel-Based Inference of Inter-Model Correspondence," in *Proc. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2007, pp. 71–80.

[60] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Gueheneuc, "Madmatch: Many-to-many approximate diagram matching for design comparison," *IEEE Trans. Software Engineering (TSE)*, vol. 39, no. 8, pp. 1090–1111, 2013.

[61] J. Rubin and M. Chechik, "Combining Related Products into Product Lines," in *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2012, pp. 285–300.

[62] T. Mende, R. Koschke, and F. Beckwermert, "An Evaluation of Code Similarity Identification for the Grow-and-Prune Model," *J. Software Maintenance and Evolution (JSME)*, vol. 21, no. 2, pp. 143–169, 2009.

[63] D. Strüber, J. Rubin, T. Arendt, M. Chechik, G. Taentzer, and J. Plöger, "RuleMerger: Automatic Construction of Variability-Based Model Transformation Rules," in *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2016, pp. 122–140.

[64] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake, "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line," in *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 316–326.

[65] D. Reuling, U. Kelter, J. Bürdek, and M. Lochau, "Automated N-Way Program Merging for Facilitating Family-Based Analyses of Variant-Rich Software," *Trans. Software Engineering and Methodology (TOSEM)*, vol. 28, no. 3, pp. 13:1–13:59, Jul. 2019.

[66] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Automatic Variation-Point Identification in Function-Block-Based Models," in *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 2010, pp. 23–32.

[67] D. Wille, Ö. Babur, L. Cleophas, C. Seidl, M. van den Brand, and I. Schaefer, "Improving Custom-Tailored Variability Mining Using Outlier and Cluster Detection," *Science of Computer Programming (SCP)*, vol. 163, pp. 62–84, 2018.

[68] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.