Software and Systems Modeling (2023) 22:1495–1517
https://doi.org/10.1007/s10270-022-01062-5

SPECIAL SECTION PAPER

RaQuN: a generic and scalable n-way model matching algorithm

Alexander Schultheiß[1] · Paul Maximilian Bittner[2] · Alexander Boll[3] · Lars Grunske[1] · Thomas Thüm[2] · Timo Kehrer[3]

Received: 16 March 2022 / Revised: 8 August 2022 / Accepted: 11 October 2022 / Published online: 21 November 2022

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 origins and model types. 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.

Keywords Model-driven engineering · n-Way model matching · Clone-and-own development · Software product lines

1 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, finding the correct location for a patch application is a non-trivial task in model patching [5], which might be done more precisely using n-way matching. Matching algorithms could also be used to find the correct location for the application of patches when synchronizing multiple variants in clone-and-own development [6]. Here, a matching-based patching technique might be more suitable than the context-based techniques implemented in version control systems [7], as variants have deliberate differences that make it difficult to find a fitting context. Lastly, matching algorithms are an indispensable basis for merging parallel lines of development [8], or for consolidating individual views to gain a unified perspective of a multi-view system [9].

Currently, almost all existing matching algorithms can only process *two* development artifacts [10–21], whereas the

Communicated by Shiva Nejati and Daniel Varro.

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

✉ Alexander Schultheiß
alexander.schultheiss@informatik.hu-berlin.de

✉ Alexander Boll
alexander.boll@inf.unibe.ch

✉ Timo Kehrer
timo.kehrer@inf.unibe.ch

Paul Maximilian Bittner
paul.bittner@uni-ulm.de

Lars Grunske
grunske@informatik.hu-berlin.de

Thomas Thüm
thomas.thuem@uni-ulm.de

[1] Humboldt-Universität zu Berlin, Berlin, Germany

[2] University of Ulm, Ulm, Germany

[3] University of Bern, Bern, Switzerland

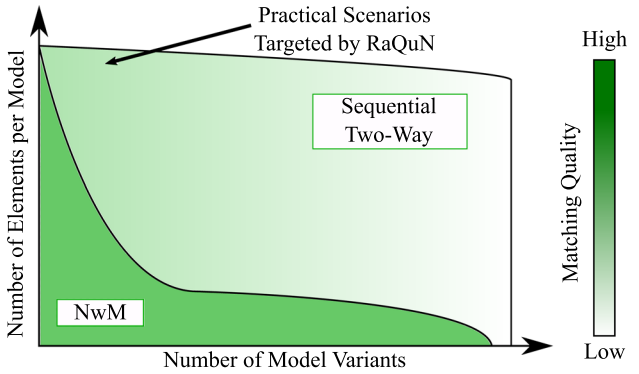

Fig. 1 Symbolic illustration of the limitations of existing n-way matching solutions and the need for further research.

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 [22–27]. In each step, the resulting two-way correspondences are simply linked together to form *correspondence groups* or *matches* (aka. *tuples* [28]).

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 [28]. 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 [9,22,29–33]. An optimal processing order cannot be anticipated and applying all $n!$ possible orders for n input models is clearly not feasible [24].

The only matching approach which simultaneously processes n input models is a heuristic algorithm called NwM by Rubin and Chechik [28]. NwM delivers n-way matchings of better quality than sequential two-way matching. However, 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 required number of model element comparisons, which often leads to performance problems even in the case of few input models if these models are large [34–36].

The limitations of existing solutions are symbolically illustrated in Fig. 1.

By applying sequential two-way matching, n-way matching can be done for both large models and large sets of models, but scalability comes at the price of quality. NwM delivers n-way matchings of better quality, but does not scale for large models, even if the number of model variants is limited to only a few. Thus, there is a strong need for a scalable n-way matching solution.

In our MODELS paper [37], we proposed RaQuN (**Ra**nge **Qu**eries on **N** input models), a generic, heuristic n-way model matching algorithm. As illustrated in Fig. 1, RaQuN targets practical scenarios in which several models are to be matched that each comprises a large number of elements. 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, RaQuN can reduce the number of required comparisons considerably. For our empirical assessment, we used datasets from different domains and development scenarios. Next to academic and synthetic models [28,38], we investigated variants generated from model-based product lines [39–42], and reverse-engineered models from clone-and-own development [1,43]. Our evaluation showed 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 this paper, we extend our previous publication [37] in three aspects. First, we evaluate RaQuN on five additional experimental subjects comprising Simulink models, a model type which we have not considered before. Second, we propose two additional configuration options for RaQuN's configuration points (cf. Sect. 4); the first option targets the reduction of RaQuN's runtime (cf. Sect. 4.1), and the second option the improvement of RaQuN's matching quality with respect to precision and recall (cf. Sect. 4.3). Finally, we extend our evaluation of the impact of RaQuN's configuration points on runtime and matching quality in terms of two additional research questions (cf. RQ1 and RQ3 in Sect. 5).

In summary, our contributions are:

Generic Matching Algorithm (Sect. 3). We present a generic simultaneous n-way model matching algorithm, RaQuN, that uses multi-dimensional search trees to find suitable match candidates.

Domain-agnostic Configuration (Sect. 4). 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 (Sect. 5). 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 current state-of-the-art approaches.

2 n-Way model 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. As our running example, we consider the three UML class diagrams A, B, and C given in Fig. 2, which are fragments of the hospital case study [28,38]. 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* [28]. 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 state charts) 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. 2, 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* [28]). We call such a 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 [22–28], 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

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

where $\mu(t)$ denotes the set of input models from which the elements of t originate. The intuitive matches illustrated in Fig. 2, 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 $(\prod_{i=1}^{n}(m_i + 1)) - 1$ [28], where n denotes the number of models, and m_i denotes the number of elements in the $i\text{-}th$ model.

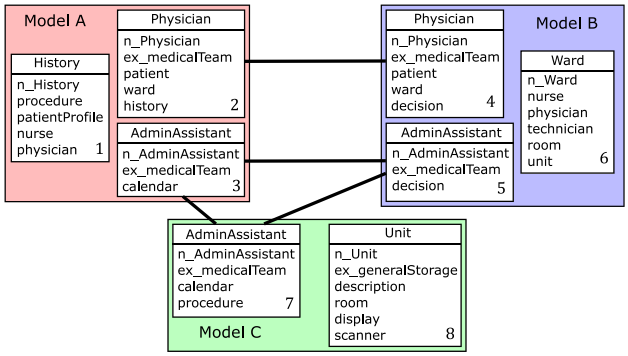

Fig. 2 Three UML models representing early design variants of the data model of a medical information system, serving as running example

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 [11,12,34–36] (cf. related work in Sect. 6) 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 [44].

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 to (i) establish a partial order on a set of matches and (ii) 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 [28] (see Sect. 4.3).

3 Generic matching algorithm

In this section, we first describe our generic n-way matching algorithm RaQuN (Algorithm 1), followed by an illustration applying the algorithm to our running example introduced in Sect. 2, 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 Sect. 4.

Algorithm 1 RaQuN

```
1:  procedure RaQuN(M)                                   ▷ A set of input models
2:      E ← ⋃_{i=1}^{i=N} M_i                            ▷ Phase 1:
3:      tree ← createEmptyTree()                           Candidate
4:      for e ∈ E do                                       Initialization
5:          v_e ← vectorize(e)
6:          tree ← insert(tree, e, v_e)
7:      end for
8:      P ← ∅                                            ▷ Phase 2:
9:      for e ∈ E do                                       Candidate
10:         Nbrs ← neighborSearch(tree, e)                 Search
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:
19:     T ← {{e} | e ∈ E}                                  Matching
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
```

3.1 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).

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, as determined by 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 [45]. For every neighbor nbr ∈ Nbrs of e, RaQuN creates an unordered pair $p = \{e,$ nbr$\}$. If p is a valid match according to Equation 1 (i.e., the two elements belong to different models), p is added to the match candidates P.

Matching (Line 18–27) In the third and last 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 pair $p \in \hat{P}$, $p = \{e, e'\}$, 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 $\hat{t} = t \cup t'$ 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 $\hat{t}$. 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.

3.2 Exemplary illustration

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

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). Figure 3 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 models 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. 3. 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

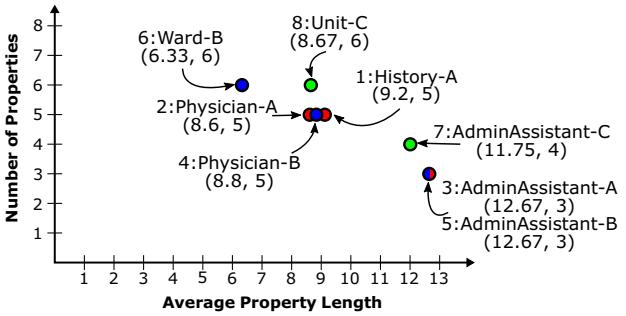

Fig. 3 Model elements of our running example mapped to points in a k-dimensional vector space (with k=2)

all elements, we obtain the set P of candidate pairs:

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

Matching s RaQuN sorts the match candidates P by descending confidence whether their elements should be matched, according to its similarity function. For the sake of illustration, we choose a straightforward similarity function: the ratio of shared properties to all properties in the two elements—known as the Jaccard Index [46]. 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, RaQuN selects 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 $\frac{3}{4} > \frac{1}{2}$. RaQuN thus merges the matches to the new match $\hat{t} = \{3, 7\}$. RaQuN replaces 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. 2.

3.3 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 $O(nm)$ [45], 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 $O(\log nm)$ [45,47]. 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 the 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., $|t \cup t'| \leq n$. 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 $\mathcal{O}(n^4m^4)$ [28]. 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, while the estimation of the worst-case complexity assumes no reduction. 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 $\hat{P}$.

4 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. In the following, we discuss possible adjustments and implementations for the different variation points in each phase of RaQuN.

4.1 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 [45].

The vectorization function defines the abstraction of model elements and their properties. It embodies RaQuN's core trade-off between runtime performance and matching quality, as it directly impacts which match candidates are retrieved and the computational effort of retrieval. Generally speaking, a vectorization function should cluster similar elements in the same region of the vector space. If more dimensions are used for vectorization, the level of abstraction is lower and the clustering of similar elements is improved, but the nearest neighbor search on the tree requires more time. If less dimensions are used, the level of abstraction is greater, which can reduce the time required to find match candidates significantly, but it also becomes less likely that suitable match candidates can be found among the neighbors in the vector space. This could negatively affect the quality of the matching, as more incorrect or missing matches might be produced.

In the following, we discuss two examples of possible vectorization functions: a low-dimensional vectorization (Low Dim) and a high-dimensional vectorization (High Dim). These are two concrete suggestions which can be applied to any element/property representation of a model; Low Dim is in favor of performance and High Dim is in favor of matching quality.

Low Dim The low-dimensional vectorization reuses the two dimensions of the very simple vectorization presented in Sect. 3.2. These two dimensions encode the average number of characters in an element's properties, and its total number of properties. Additionally, for each unique character in an element's properties, there is one dimension that represents the number of occurrences of that character in the element's properties. The number of dimensions is bound by the size of the alphabet and may be reduced by omitting those dimensions which represent characters that do not occur in any property name.

High Dim The high-dimensional vectorization 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. Thus, the vectorization performs a one-hot encoding of all distinct properties in the input models. 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. The number of required dimensions dynamically grows with the number of distinct properties and thus with number and size of input models.

4.2 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 Sect. 3.2. 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.

4.3 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 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 relying only on the generic element-property representations when implementing *shouldMatch*.

The *similarity* function, which determines the similarity of elements, is applied to assess the quality of a matching as illustrated in Sect. 2. It is used to sort the match candidates $\hat{P}$ in Line 18 such that more similar pairs are considered to be merged first. In the following, we discuss two possible similarity functions and their corresponding *shouldMatch* predicate.

Weight Metric The first similarity function is the *weight* metric by Rubin and Chechik [28], 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

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

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 the match t.

For the configuration of *shouldMatch*[1] we follow the match decision proposed by Rubin and Chechik [28]. The idea is that any extension of a match should increase the quality of the overall matching. 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:

$$shouldMatch_w(t, t', e, e') := w(t \cup t') > w(t) + w(t') \quad (3)$$

Jaccard Index The second similarity function is the Jaccard Index, which we applied in our motivating example in Sect. 3.2. The Jaccard Index is a wide-spread similarity metric for sets; it is named after Paul Jaccard who first defined it as *coefficient de communauté* in his work on flora in the alpine zone [46]. The Jaccard Index can be applied to our matching problem, because elements are sets of properties. We want to match elements that have many common properties, while having only few individual properties. Given a match t, the Jaccard index is calculated as

$$J(t) = \frac{\bigcap_{e \in t} e}{\bigcup_{e \in t} e} \quad (4)$$

where e is an element, which we consider to be a set of properties. A greater Jaccard Index index corresponds to greater similarity.

Regarding *shouldMatch*, we define that elements should be matched if their similarity is greater or equal to a predefined similarity threshold s:

$$shouldMatch_J(t, t', e, e') := J(t \cup t') \geq s \quad (5)$$

While *shouldMatch* used by the weight metric matches two elements greedily (i.e., elements can be matched if they have at least one common property), a threshold-based definition allows the specification of the desired minimum similarity of matches. Thereby, it is also possible that extending a match might decrease its match quality, as long as the minimum similarity is kept. It is not feasible to use a similarity threshold for the weight metric, because the weight also depends on the size of the match and the number of considered models; different thresholds would have to be applied depending on the considered models and the current match size.

We expect that each configuration option presented in this section has an impact on RaQuN's runtime and matching quality. We want to investigate this impact empirically.

5 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 [48] and GitHub[2].

RQ1 How does the configuration of RaQuN's candidate initialization (i.e., vectorization) affect its matching quality and runtime?
RQ2 Is $k' = n$ a suitable heuristic for the number of considered neighbors during RaQuN's candidate search?
RQ3 How does the configuration of RaQuN's candidate matching affect its matching quality?
RQ4 How does RaQuN perform compared to NwM and sequential two-way matching in terms of matching quality and runtime?
RQ5 How does RaQuN scale with growing model sizes?

5.1 Selected algorithms

Table 1 summarizes the matchers we used for our experiments. We compare different configurations of RaQuN, NwM, and two sequential two-way approaches (Pairwise). All matchers are implemented in Java.

5.1.1 Prototypical implementation of RaQuN

We implemented a prototype of RaQuN which uses a generic kd-tree library by the Savarese Software Research Corporation [49]. For all other variation points, we implemented the domain-agnostic configuration options discussed in Sect. 4 as extension of the prototype. First, for the comparison of vectorization functions (cf.Sect. 4.1), we implemented RaQuN Low Dim and RaQuN High Dim named according to their vectorization function; both use the weight metric as similarity function. Second, for the comparison of similarity functions (cf.Sect. 4.3), we additionally implemented RaQuN Jaccard using the Low Dim vectorization function and the Jaccard Index as similarity function.

[1] The *shouldMatch* predicate takes the two matches, t and t', and the elements in the candidate pair, e and e', as input and returns $true$ or $false$. While e and e' are not used by the *shouldMatch* predicates presented here, we extended the predicate's interface to allow for match decisions that explicitly take e and e' into account.

[2] https://github.com/AlexanderSchultheiss/RaQuN.

Table 1 Selected algorithms and their configurations

Name	Type	Vectorization F.	Similarity F.
RaQuN Low Dim	n-way	Low Dim	Weight
RaQuN High Dim	n-way	High Dim	Weight
RaQuN Jaccard	n-way	Low Dim	Jaccard Index
NwM	n-way	N/A	Weight
Pairwise Ascending	two-way	N/A	Weight
Pairwise Descending	two-way	N/A	Weight

5.1.2 Baseline algorithms

All baseline matchers use the weight metric [28], defined in Equation 2, as similarity function (see Sect. 2). The prevalent way to calculate n-way matchings is sequential two-way [22–27]. 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 [50] to maximize the weight of the matching in each iteration. For (b), Rubin and Chechik [28] 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 [28].

5.2 Experimental subjects

Our experimental subjects and their basic characteristics are summarized in Table 2.

5.2.1 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 [28]. 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 [38]. 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.

5.2.2 Variants generated from 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. [51].

The Pick and Place Unit (*PPU*) is a laboratory plant from the domain of industrial automation systems [52,53] whose system structure and behavior are described in terms of SysML block diagrams and UML statemachines, respectively [39]. Variation arises from different scenarios supported by the plant. The Barbados Car Crash Crisis Management System (*bCMS*) [40,54] 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 [40], including both functional and non-functional variability. The Body Comfort System (*BCS*) [41] 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. [41]. *ArgoUML* is a publicly available CASE-tool supporting model-driven engineering with the UML. It was used in prior studies [55,56] 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 [42]. They represent different tool variants which have been extracted by removing specific features for supporting different UML diagrams.

5.2.3 Variant sets created through clone-and-own

Another subject stems from a software family called *ApoGames* which has been developed using the clone-and-own approach [1,43] (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 [57]. The challenge comprises 20 Java and five Android variants, from which we selected the Java variants only.

5.2.4 Simulink subjects

We also included five Simulink subjects. Three of them are case studies taken from Schlie et al. [58,59]: DAS, a driver assistance system from the SPES_XT project [60], and APS,

Table 2 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
Apo-Games	Simple class diag.	20	63.05	60	19.62	13
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
DAS	Simulink	19	842.37	879	11.00	11
APS	Simulink	7	202.71	206	11.00	11
APS-TL	Simulink	5	181.20	165	11.00	11
MRC	Simulink	3	773.33	970	11.00	11
WEC	Simulink	6	650.00	668	11.00	11

and APS_TL, an auto platooning system from the CrEst project [61]. Schlie et al. extracted *module building blocks* from the Simulink models, which he then recombined in different combinations to generate variants of the three systems. We followed his process to generate 19, 7, and 5 variance models, respectively.

As Boll et al. previously found open-source Simulink models to be suitable for empirical research [62], we mined GitHub for open-source projects with Simulink models. We found 317 distinct projects with 4,402 Simulink models. In this set, we conducted a basic search for Simulink model "twins", by looking for models with identical qualified names, *i.e.* having the same subdirectory path and file name. Our intention of this was finding variations of Simulink models in different forks. We view these forked Simulink models as a substitute for variants. To this end, we investigated the "twins" and rejected identical models (by hashsum and then manual inspection), models of trivial size, and models without any matchable elements. This search and filtering yielded two families: MRC comprising three[3] and WEC comprising six Simulink models.[4]

5.2.5 Generation of ArgoUML subsets

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.

To that end, in addition to the presented experimental subjects, we generated subsets of ArgoUML, which comprises the largest models of our subjects. The subjects are presented in Table 3.

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 the selection 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.

[3] These stem from the MRC contest (https://de.mathworks.com/matlabcentral/fileexchange/50227-mission-on-mars-robot-challenge-2015-france). Contestants constructed variants of a robot that identifies obstacles and avoids them.

[4] Here, forks modified a library model stemming from the open-source wave energy conversion simulator (WEC) (https://wec-sim.github.io/WEC-Sim/master/index.html).

Table 3 ArgoUML subsets and their characteristics

	Size	Elements		Properties	
		Avg.	Median	Avg.	Median
Argo-Subset-1	1%	18.52	19	8.88	4
Argo-Subset-5	5%	93.83	94	8.61	4
Argo-Subset-10	10%	187.16	187	8.57	4
Argo-Subset-15	15%	278.39	278	8.51	4
Argo-Subset-20	20%	369.08	369	8.61	4
Argo-Subset-25	25%	459.68	460	8.59	4
Argo-Subset-30	30%	549.15	549	8.53	4
Argo-Subset-35	35%	637.57	638	8.58	4
Argo-Subset-40	40%	726.04	725	8.49	4
Argo-Subset-45	45%	813.00	813	8.54	4
Argo-Subset-50	50%	900.15	899	8.54	4
Argo-Subset-55	55%	987.39	987	8.60	4
Argo-Subset-60	60%	1073.47	1073	8.59	4
Argo-Subset-65	65%	1159.51	1159	8.60	4
Argo-Subset-70	70%	1245.00	1244	8.61	4
Argo-Subset-75	75%	1330.52	1329	8.61	4
Argo-Subset-80	80%	1415.41	1416	8.64	4
Argo-Subset-85	85%	1499.65	1496	8.65	4
Argo-Subset-90	90%	1584.42	1583	8.67	4
Argo-Subset-95	95%	1668.24	1665	8.68	4
ArgoUML	100%	1752.86	1749	9.05	4

5.2.6 Conversion to element/property representations

Converting the experimental subjects into element/property representations requires a pre-processing step that is model-type and technology-specific. The main idea is to convert those entities of a model into elements, for which a match should be found, and to convert all entities that are related to an element into its properties. Here, a domain expert decides which properties are (possibly) relevant for distinguishing elements (e.g., an element's position in a visual representation might not be relevant). For example, if UML activity diagrams are to be matched, each activity could be converted into an element, and the properties of an activity could be its name, as well as the names of its preceding and subsequent activities. In this case, an element's property (i.e., its name) is also a property of another element. We followed this idea for the conversions of our experimental subjects.

For class diagrams, we convert classes and interfaces to elements; the properties are the class' or interface's name, its method signatures, and names of fields. For statemachines, we convert states and transitions into elements; the properties of a state are the names of its incoming and outgoing transitions, as well as the names of its actions; the properties of a transition are the names of its source and target state, as well as the names of its effect and guard. For SysML block diagrams, we convert blocks into elements; the properties are the names of a block's attributes. For component/connector diagrams, we convert components and connectors into elements; the properties of a component are the names of its ports, as well as the names of incoming and outgoing connectors; the properties of a connector are the name of its type, as well as the names of its source and target component. Lastly, for Simulink models, we convert Simulink blocks into elements; the properties are similar to the ones considered by Schlie [59]: The name of a block, the block's type, the name of its parent, the numbers of its inputs and outputs, and its graphical position.

For the conversions implementation, we used the generic EMF model traversal and reflective API to access an element's local properties and referenced elements. Elements and properties of the Simulink models were accessed via basic Simulink getter routines, as well. Our pre-processing code is part of our replication package [48].

5.3 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 [51]. We use the two most widely established quality evaluation metrics *weight* and *precision/recall*.

5.3.1 Weight

One way to measure the quality of an n-way matching is the weight metric [28], which we also use as a similarity function (cf. Sect. 4.3), 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 Equation 2. There can be several matchings with the same weight, and thus several optimal matchings for a set of models. We chose the weight metric as it does not depend on a ground truth, which is often not available.

5.3.2 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 [63]. For our experimental subjects, however, such oracles are only available for models generated from a software product line. 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 a product line (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:

$$TP(T) := \sum_{t \in T} \left| \left\{ \{e_1, e_2\} \subseteq t \mid id(e_1) = id(e_2) \right\} \right| \tag{6}$$

$$FP(T) := \sum_{t \in T} \left| \left\{ \{e_1, e_2\} \subseteq t \mid id(e_1) \neq id(e_2) \right\} \right| \tag{7}$$

$$FN(T) := \left| \bigcup_{\substack{t_1, t_2 \in T \\ t_1 \neq t_2}} \left\{ \{e_1, e_2\} \mid e_1 \in t_1, e_2 \in t_2, id(e_1) = id(e_2) \right\} \right| \tag{8}$$

Precision, recall, and F-measure are calculated as usual [63]:

$$\text{precision(T)} = \frac{|TP(T)|}{|TP(T)| + |FP(T)|}$$

$$\text{recall(T)} = \frac{|TP(T)|}{|TP(T)| + |FN(T)|}$$

$$\text{F-measure(T)} = 2 \cdot \frac{precision(T) \cdot recall(T)}{precision(T) + recall(T)}$$

Precision expresses how many formed matches are correct, recall expresses how many required matches have been formed, and F-measure is the harmonic mean of precision and recall.

5.4 Methodology and results

We ran 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 matcher 30 times on each of our experimental subjects, except for Random, Loose, and Tight for which we follow the methodology of Rubin and Chechik [28]. Here, we select 10 subsets comprising 10 of the 100 models for each run that is repeated 30 times, leading to 300 runs per matcher and subject. Regardless of the experimental subject, we permutate the input models randomly for each experimental run to minimize the potential impact that the order of models might have on the result, due to RaQuN's *filterAndSort* (cf. Line 18) not determining a fixed order in the case of candidate pairs having equal similarity. We set a time-out of 12 hours for each run, due to the high number of experimental runs.

For our time measurements, we do not consider the time required for the conversion of the models to an element/property representation, because it is a one-time preprocessing step that is detached from n-way matching.

For smaller subjects (e.g., PPU Structure), the conversion took less than one second. For the larger datasets (e.g., PPU Behavior, ArgoUML), the conversion took less than five seconds. Furthermore, the conversion time is dominated by the time required for IO operations. Thus, the hardware on which the models are stored (i.e., secondary storage device) is a main factor.

5.4.1 RQ1: configuration of the candidate initialization

To assess how the configuration of the candidate initialization (i.e., which vectorization function is used) impacts RaQuN's performance, we compare RaQuN Low Dim and RaQuN High Dim on the experimental subjects presented in Sect. 5.2. Table 4 presents the average weight and runtime achieved by the two configurations.

Here, we consider the weight metric as it does not require a ground truth, which is not available for all datasets.

With respect to runtime, both configurations can compute a matching for each of the experimental subjects, requiring at most a couple of minutes for all subjects besides ArgoUML. RaQuN Low Dim is significantly faster than RaQuN High Dim across all datasets; its smallest *relative* speed-up of a factor of 1.6 can be observed on DAS, and the largest relative speed-up of a factor of 71.0 on ArgoUML. In terms of *absolute* runtime differences, RaQuN Low Dim achieves only a minor advantage on small datasets (e.g., Hospital, Warehouse, or Random), but it can compute a matching for ArgoUML – the experimental subject with the largest models – in less than one minute, while RaQuN High Dim requires almost an hour. With respect to weight, both configurations achieve similar matching weight on the majority of experimental subjects, but RaQuN High Dim computes matchings with higher weight on almost all experimental subjects.

These results show, that both of our generic vectorization functions lead to varying results, depending on the characteristics of the experimental subjects (cf.Table 2). This is not surprising, because mapping elements to points in the vector space based on property names (High Dim), or based on the characters used in the properties' names (Low Dim), is directly affected by the characteristics of the subject. The only subject, on which RaQuN High Dim computes a matching with slightly lower weight, is DAS. DAS is also the subject on which the smallest relative runtime difference was measured, which suggests that, for DAS, both vectorization functions lead to a similar mapping of elements to points in the vector space.

Table 4 Comparison of achieved weights and runtimes across RaQuN configurations, 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 High Dim.	**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]
RaQuN Low Dim.	4.04	**0.01** [0.01, 0.03]	1.53	**0.05** [0.04, 0.08]	0.82	**0.02** [0.01, 0.18]	0.77	**0.01** [0.01, 0.08]	0.84	**0.02** [0.01, 0.10]	**18.27**	**1.23** [1.16, 1.40]
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 High Dim.	**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]		
RaQuN Low Dim.	**28.95**	**0.10** [0.09, 0.17]	164.26	**9.61** [9.12, 10.90]	41.47	**0.22** [0.20, 0.32]	51.16	**5.75** [5.45, 6.72]	1727.45	**37.71** [33.56, 45.65]		
Algorithm	DAS		APS		APS-TL		MRC		WEC			
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)		
RaQuN High Dim.	732.27	1,073.55 [943.73, 1, 272.92]	**169.41**	3.74 [2.06, 7.26]	**153.72**	1.78 [0.78, 5.55]	**534.22**	51.39 [31.97, 94.79]	**263.14**	269.11 [201.61, 536.52]		
RaQuN Low Dim.	**732.31**	**669.67** [644.30, 748.11]	**169.41**	**1.41** [1.32, 1.56]	**153.72**	**0.36** [0.33, 0.52]	530.16	**1.01** [0.87, 1.26]	257.97	**5.13** [3.85, 6.20]		

Bold values correspond to the highest weight and lowest runtime in each column

Both generic configurations, RaQuN Low Dim and RaQuN High Dim, achieve their intended purpose and make it feasible to compute a matching for each experimental subject. The configuration of the candidate initialization has a significant impact on the runtime and match quality. While there is a noticeable trade-off between runtime and weight, opting for the maximization of weight is reasonable for almost all subjects.

5.4.2 RQ2: suitability of k' heuristic during candidate search

In order to assess the suitability of $k' = n$ as heuristic for the number of neighbors during the candidate search, we ran RaQuN Low Dim and RaQuN High Dim with increasing values of k'. We observed highly similar results for RaQuN Low Dim and RaQuN High Dim, and we thus discuss the results for RaQuN High Dim in the remainder of this section to reduce redundancy.

Figure 4 presents the results of the runs of RaQuN High Dim 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 required to reach RaQuN's peak weight performance. 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'. There are several reasons for this. First, multiple elements can be mapped to the same point in the vector space, leading to more than k' elements being retrieved by the candidate search (cf. Sect. 3.2). Second, the candidate search is performed for each element. In our examplary illustration (cf. Sect. 3.2), RaQuN retrieves only one match candidate (*4:Physician-B*) for *1:History-A*, but *1:History-A* is part of three candidate pairs, because it is retrieved as match candidate for two other elements (*5:AdminAssistant-A* and *7:AdminAssistant-C*). Third, elements can be matched transitively, if they have a common match candidate. This is because the matching phase merges matches that contain the match candidates, if *shouldMatch* evaluates to *true* (cf. Sect. 3.1). For example, an element A is a candidate for an element B, and B is a candidate for an element C; if A and B are matched, RaQuN might add C to the match, after considering the candidate pair containing B and C.

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 consid-

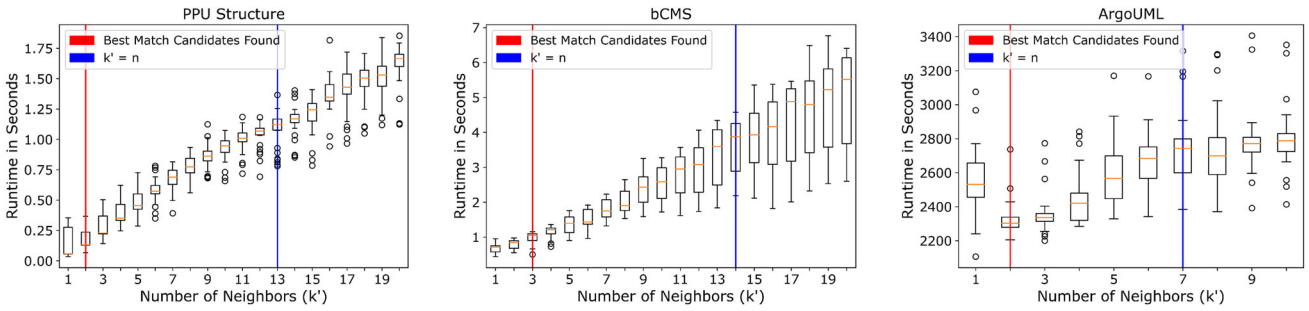

Fig. 4 Impact of an increasing number of neighbors considered for matching on the performance of RaQuN High Dim

ering more neighbors than necessary will not cause a sudden increase in runtime.

Table 5 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 of 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.

5.4.3 RQ3: configuration of RaQuN's candidate matching

RaQuN's third configuration aspect is the similarity function and its *shouldMatch* predicate (cf.Sect. 4.3). We compare the weight metric and the Jaccard Index as two possible options for the configuration of RaQuN's matching phase (cf.Sect. 4.3). For the Jaccard Index, we also have to set a value for the similarity threshold of its *shouldMatch* predicate (cf.Sect. 4.3). In practice, we envision that the similarity threshold is customized with respect to the similarity required for subsequent development activities. However, for the sake of evaluating RaQuN, we consider model matching independent of subsequent activities. Instead, we evaluate the Jaccard Index with a range of similarity thresholds from 0.25 through 1.00 in steps of 0.25.

Furthermore, while using the weight metric to assess the quality of matches is valid when comparing matchers that all rely on the same similarity function, it suffers from a bias when comparing different similarity functions. More specifically, we cannot conduct a comparison of matchers using the weight metric and matchers using the Jaccard Index as shown in Table 4, because using the weight metric for evaluation would favor matchers that internally use the weight metric to decide whether elements should be matched. Therefore, we answer the research question by matching our ArgoUML subsets. The subsets comprise unique identifiers making it possible to calculate precision, recall, and F-measure (cf.Sect. 5.3), which we consider to be unbiased evaluation metrics for the comparison of different similarity functions. Figure 5 presents the average precision, recall, and F-measure of RaQuN Low Dim using the weight metric, and RaQuN Jaccard using the Jaccard Index.[5]

First, when considering precision (i.e., how many formed matches are correct) achieved by RaQuN Low Dim and RaQuN Jaccard, we observe that the precision of all matchers increases with increasing subset size. This is because the subsets are generated randomly by removing elements from the models. In turn, elements in the smaller subsets have fewer corresponding elements in other models, which increases the chance of matching elements that should not be matched. Furthermore, we observe differences between the precision of the matchers, depending on the similarity threshold of the Jaccard Index: Generally speaking, a higher threshold leads to higher precision. We expected this result because the likelihood of a match being correct correlates with the similarity of its elements. This is also the reason why RaQuN Low Dim achieves similar precision as RaQuN Jaccard with a threshold of 0.25; the weight metric forms matches greedily (i.e., elements can be matched if they have at least one common property), which is comparable to a small similarity threshold.

Second, with respect to recall (i.e., have all required matches been formed), we observe almost no difference between the two similarity functions. RaQuN Low Dim and RaQuN Jaccard achieve a high recall between 0.95 and 1.00 across all datasets. Only RaQuN Jaccard using a similarity threshold of 1.00 achieves a significantly lower recall across all subsets. This is not surprising, because a threshold of 1.00 only matches elements that have exactly the same set of properties, while a match can also be correct if the elements have a few different properties.

[5] Both configurations use the Low Dim vectorization to reduce the runtime of the experiment.

Table 5 Number of element comparisons that are saved by RaQuN High Dim with $k'=n$

Dataset	Full n-way matching #Comparisons	RaQuN #Comparisons	Saved
Hospital	21, 211	936	95.6%
Warehouse	70, 037	4044	94.2%
Random	27, 918	1964	93.0%
Loose	26, 716	1995	92.5%
Tight	28, 982	1569	94.6%
Apo-Games	750, 319	31, 028	95.9%
PPU Structure	80, 620	30, 853	61.7%
PPU Behavior	3, 814, 644	207, 385	94.6%
bCMS	416, 571	44, 336	89.4%
BCS	939, 346	164, 860	82.4%
ArgoUML	64, 521, 622	362, 890	99.4%
DAS	121, 115, 254	3, 700, 634	96.9%
APS	858, 294	33, 500	96.1%
APS-TL	325, 143	9584	97.1%
MRC	1, 675, 724	6745	99.6%
WEC	6, 128, 714	24, 306	99.6%

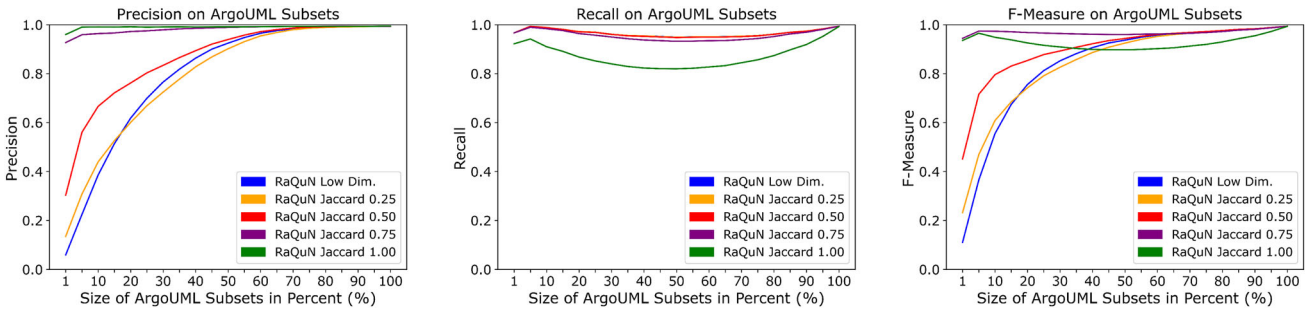

Fig. 5 Average precision, recall and F-measure of RaQuN Low Dim and RaQuN Jaccard on subsets of ArgoUML with increasing size. RaQuN Jaccard was run with varying thresholds for its *shouldMatch* predicate, ranging from 0.25 to 1.00

Finally, with respect to F-measure (i.e., the harmonic mean between precision and recall), the results show that RaQuN Jaccard using a similarity threshold of 0.75 achieved the best matching quality across all subsets and that RaQuN Low Dim and RaQuN Jaccard with thresholds of 0.25 and 1.00 achieved the worst overall matching quality depending on the subset size.

RaQuN Jaccard achieves similar or better matching quality than RaQuN Low Dim, depending on the chosen similarity threshold. By considering a range of thresholds, we found that RaQuN Jaccard with a high threshold (i.e., greater than 0.75) can achieve almost perfect precision, but that selecting a too high threshold (i.e., 1.00) negatively affects the recall. In practice, RaQuN should apply a similarity function with a similarity threshold in order to reduce the number of incorrect matches.

5.4.4 RQ4: comparison with other algorithms

For the comparison of RaQuN against the baseline matchers NwM, Pairwise Ascending, and Pairwise Descending, we assess the differences in average runtime and match weight on each experimental subject. We further evaluate the quality of matchings in terms of precision, recall, and F-measure on ArgoUML subsets. Based on our earlier conclusion that using the high-dimensional vectorization is the preferable choice (cf. Sect. 5.4.1), we consider RaQuN High Dim as representative of RaQuN. RaQuN High Dim uses the weight metric as similarity function, because the baseline matchers also use the weight metric.

Table 6 presents the average weight and runtime achieved by the matchers.

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

Table 6 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 High Dim.	**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 [3742.22, 6597.83]
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 High Dim.	**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 [1483.48, 3324.70]
NwM	28.64	9.27 [7.84, 10.07]	146.35	4,616.30 [3410.86, 5919.65]	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]

Algorithm	DAS		APS		APS-TL		MRC		WEC	
	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)	Weight	Time (in s)
RaQuN High Dim.	**732.27**	1,073.55 [943.73, 1, 272.92]	**169.41**	**3.74** [2.06, 7.26]	**153.72**	**1.78** [0.78, 5.55]	**534.22**	51.39 [31.97, 94.79]	**263.14**	269.11 [201.61, 536.52]
NwM	- - - timeout - - -		- - - timeout - - -		- - - timeout - - -		- - - timeout - - -		- - - timeout - - -	
Pairwise Ascending	683.75	953.61 [774.91, 1, 712.88]	148.62	7.46 [4.73, 11.66]	136.02	3.69 [1.91, 9.16]	532.76	23.97 [13.63, 37.63]	253.47	65.37 [39.25, 110.43]
Pairwise Descending	730.40	**809.85** [757.50, 1, 213.46]	165.28	4.93 [4.78, 5.80]	152.23	3.12 [1.99, 4.92]	534.18	**14.97** [13.70, 17.34]	258.05	**46.92** [39.91, 73.30]

Bold values correspond to the highest weight and lowest runtime in each column

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, DAS, APS, APS-TL, MRC, and WEC before reaching the time-out of 12h, and it took about 90min and 70min for matching Apo-Games and PPU Behavior, respectively. In contrast, RaQuN provides a matching in an average time of less than 45 minutes for ArgoUML, less than 20 minutes for DAS, 71s for Apo-Games, and less than 5 minutes for the remaining subjects.

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 [28]. Moreover, which of the two Pairwise matchers performs better changes from subject to subject, making it not possible to anticipate which order will yield better results.

The comparison of matching quality in terms of precision, recall, and F-measure is presented in Fig. 6.

First, the precision achieved by the different algorithms is presented in the leftmost plot of Fig. 6; for NwM, we only have results for subsets with a size of up to 40%, because the timeout of 12 hours was reached for larger subsets. 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.

Second, the central plot of Fig. 6 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 subset 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, lead to a higher loss in recall.

Lastly, the rightmost plot of Fig. 6 presents the F-measure achieved by the matchers. Here, we observe that RaQuN offers the best trade-off between precision and recall across all ArgoUML subsets.

RaQuN High Dim is significantly faster than NwM on all subsets, and is almost as fast as two-way matchers on medium-sized subsets with hundreds of elements. RaQuN High Dim achieves the highest weights across all subsets and is able to deliver matchings with the highest precision and recall on the ArgoUML subsets.

5.4.5 RQ5: scalability with growing input size

As opposed to NwM, RaQuN shows great scaling properties for models of increasing size, up to the largest models of our subjects containing more than 10,000 elements in total. By using a vectorization function that opts for better runtime, RaQuN Low Dim computes matches even faster than Pairwise. This is a strong indicator that RaQuN's typical scaling behavior is considerably better than its theoretical worst-case complexity.

The results of our scalability analysis on the ArgoUML subsets is shown in Fig. 7, which 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. RaQuN and Pairwise show similar scaling properties, while RaQuN's absolute runtime depends on the used vectorization function. For matching the full models (cf. Table 6), the average runtime of RaQuN High Dim was less than 45 minutes, making it slower than Pairwise, but still feasible. On the other hand, the average runtime of RaQuN Low Dim was less than one minute making it significantly faster than even the Pairwise matchers.

5.5 Threats to validity

5.5.1 Construct valdity

Our experiments rely on evaluation metrics and algorithm configurations that may affect the construct validity of the results. First, we use the weight metric which has already been used in prior studies [28]. While it can be applied to

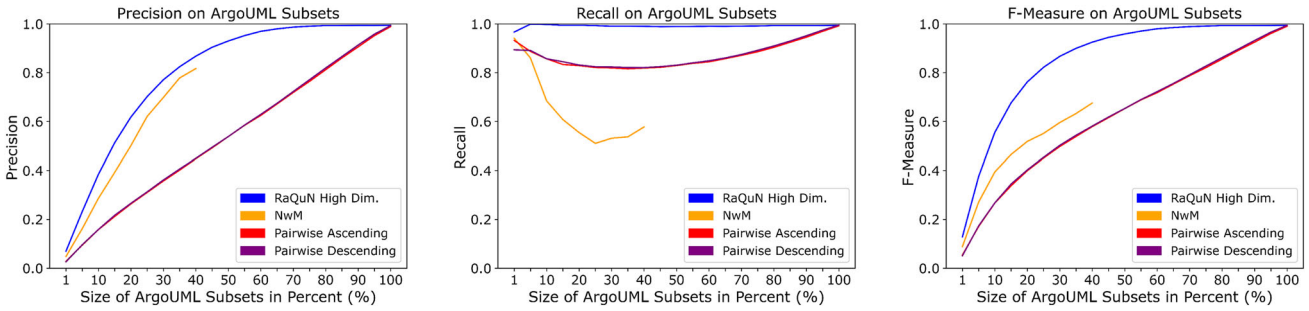

Fig. 6 Average precision, recall and F-measure of RaQuN, NwM, and Pairwise on subsets of ArgoUML with increasing size

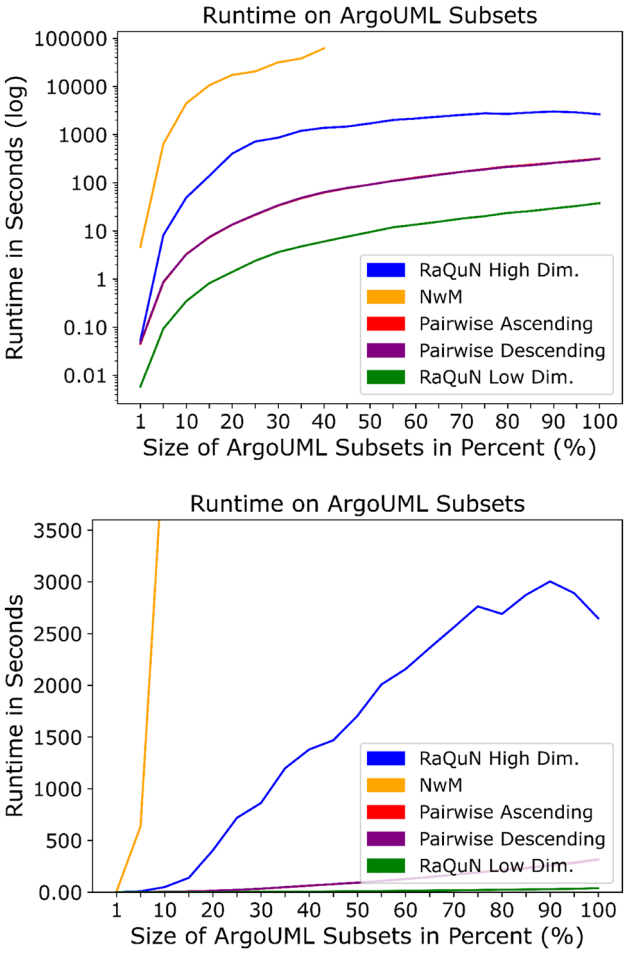

Fig. 7 Average runtime of RaQuN, NwM, and Pairwise, on subsets of ArgoUML with increasing size, in logarithmic scale (top) and linear scale (bottom). The linear plot shows only runtimes shorter than an hour

compare the results on the same subject, weights obtained for different experimental subjects are hardly comparable. To that end, we use precision and recall [63] 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 [64–66]. Thus, while it is common in the literature to use product-line datasets [55,56,67,68] as they inherently provide ground-truth matchings, we also considered the clone-and-own system ApoGames.

Lastly, regarding the configuration of RaQuN's candidate matching phase (cf. Sect. 5.4.3), we compare the weight metric and Jaccard Index using two configurations of RaQuN that use the Low Dim vectorization. We use the Low Dim vectorization mainly because we could then conduct the experiment considerably faster. This might introduce a bias to the results, as the selected candidates depend on the vectorization and different candidates could lead to different results. However, as observed in Table 4, both vectorizations lead to comparable matching quality. We thus deem our conclusion to still hold.

5.5.2 Internal valdity

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.

5.5.3 External valdity

The question whether the results generalize to other subjects, 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 [28]. 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), component/connector models (BCS), and Simulink models.

To apply RaQuN as investigated in this paper, models must first be converted to element/property models. The conversion is model-type and technology-specific, and requires domain knowledge in order to select suitable elements and properties. It also leads to the abstraction of a model's structural features (i.e., hierarchies and relationships), which might no longer yield the information required for it to be useful in the matching process. Furthermore, RaQuN does not consider already established matches or similarities of elements that are in a (structural) relationship with the elements that are to be matched next (e.g., the similarity of parent elements in a hierarchical structure). Thus , the conversion can have a negative impact on the matching process, which could lead to a reduction of the overall matching quality. This threat applies to all considered algorithms, as we evaluate them on the same element/property models. We partially mitigate this threat by our selection of various experimental subjects.

6 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 [69] - 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 [10–14,20]. Distance-based approaches try to establish a matching which yields a minimal edit distance [15–19,21]. 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 [11,12,34–36].

A few approaches realize n-way matching by the *repeated two-way matching* of the input artifacts [22–27]. However, as reported by Rubin and Chechik [28] 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 [24,28].

To the best of our knowledge, Rubin and Chechik are the only ones who have studied the *simultaneous* matching of n input models [28]. 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 Algorithm 1). Even without these phases, the matching (Phase 3) differs from NwM by abstaining from bipartite graph-matching, reducing the worst-case complexity (see Sect. 3.3).

Our usage of multi-dimensional search trees is inspired by Treude et al. [34]. 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. [70], 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 [29,43,51,68,71–76]. Anchor points may be determined through clone detection [43,72–74] or conventional matchers [29,51,71,75,76], 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 [51,73–75]. Merge refactoring approaches are complementary to our approach, because they require sufficiently accurate matchings to avoid prohibitive computational efforts during refactoring [51].

Another approach for managing cloned software variants has been presented by Linsbauer et al. [33,56]. 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. [30]. However, the required additional information such as complete feature configurations is typically not available.

Finally, Babur et al. [31,32] 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. [77], 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.

7 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 (Apo-Games). 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, we plan an in-depth investigation of RaQuN's potential for 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).

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., Czarnecki, K.: An exploratory study of cloning in industrial software product lines. In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 25–34. IEEE (2013)
2. Feldmann, S., Fuchs, J., Vogel-Heuser, B.: Modularity, variant and version management in plant automation–future challenges and state of the art. In: Proceedings of the International Design Conference (IDC), pp. 1689–1698 (2012)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, USA (2012)
4. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 7:1–7:8. ACM (2013)
5. Kehrer, T., Kelter, U., Taentzer, G.: Propagation of software model changes in the context of industrial plant automation. Automatisierungstechnik **62**(11), 803–814 (2014)
6. Kehrer, T., Thüm, T., Schultheiß, A., Bittner, P.M.: Bridging the gap between clone-and-own and software product lines. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 21–25. IEEE (2021)
7. Schultheiß, A., Bittner, P.M., Thüm, T., Kehrer, T.: Quantifying the potential to automate the synchronization of variants in clone-and-own (2022)
8. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. (TSE) **28**(5), 449–462 (2002)
9. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requir. Eng. **11**(3), 174–193 (2006)
10. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 54–65. ACM (2005)
11. Kelter, U., Wehren, J., Niere, J.: A generic difference algorithm for UML models. Softw. Eng. **P–64**, 105–116 (2005)
12. Melnik, S., Garcia-Molina, S., Rahm, E.: Similarity flooding: a versatile graph matching algorithm and its application to schema matching. In: Proceedings of the International Conference on Data Engineering (ICDE), pp. 117–128, IEEE (2002)
13. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of statecharts specifications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 54–64. IEEE (2007)
14. Brun, C., Pierantonio, A.: Model differences in the eclipse modeling framework. Eur. J. Inf. Prof. **9**(2), 29–34 (2008)
15. Miller, W., Myers, E.W.: A file comparison program. Softw. Pract. Exp. **15**(11), 1025–1040 (1985)
16. Canfora, G., Cerulo, L., Di Penta, M.: Ldiff: an enhanced line differencing tool. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 595–598. IEEE (2009)
17. Asaduzzaman, M., Roy, C.K., Schneider, K.A., Di Penta, M.: LHDiff: a language-independent hybrid approach for tracking source code lines. In: Proceedings of the International Conference on Software Maintenance (ICSM), pp. 230–239. IEEE (2013)
18. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. (TSE) **33**(11), 725–743 (2007)
19. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 333–343. IEEE (2007)
20. Apiwattanapong, T., Orso, A., Harrold, M.J.: A differencing algorithm for object-oriented programs. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 2–13, IEEE (2004)
21. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 313–324 (2014)
22. Wille, D., Schulze, S., Seidl, C., Schaefer, I.: Custom-tailored variability mining for block-based languages. In: Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 271–282. IEEE (2016)
23. Wille, D. Schulze, S., Schaefer, I.: Variability mining of state charts. In: Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD), pp. 63–73. ACM (2016)
24. Duszynski, S.: Analyzing similarity of cloned software variants using hierarchical set models. Ph.D. dissertation, University of Kaiserslautern (2015)
25. Klatt, B., Küster, M.: Improving product copy consolidation by architecture-aware difference analysis. In: Proceedings of the International Conference on Quality of Software Architectures (QoSA), pp. 117–122. ACM (2013)
26. Ryssel, U., Ploennigs, J., Kabitzsch, K.: automatic library migration for the generation of hardware-in-the-loop models. Sci. Comput. Program. (SCP) **77**(2), 83–95 (2012)
27. Schlie, A., Schulze, S., Schaefer, I.: Recovering variability information from source code of clone-and-own software systems. In: Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS), pp. 1–9. ACM (2020)
28. Rubin, J., Chechik, M.: N-way model merging. In: Proceedings of the European Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE), pp. 301–311. ACM (2013)
29. Reuling, D., Kelter, U., Ruland, S., Lochau, M.: SiMPOSE-configurable N-way program merging strategies for superimposition-based analysis of variant-rich software. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 1134–1137. IEEE (2019)
30. Assunção, W.K.G., Vergilio, S.R., Lopez-Herrejon, R.E.: Automatic extraction of product line architecture and feature models from UML class diagram variants. J. Inf. Softw. Technol. (IST) **117**, 106198 (2020)
31. Babur, Ö: Statistical analysis of large sets of models. In: Proceedings of the International Conference on Automated Software Engineering (ASE), pp. 888–891. ACM (2016)
32. Babur, Ö, Cleophas, L.: Using N-grams for the automated clustering of structural models. In: Proceedings of the Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). pp. 510–524, Springer (2017)
33. Fischer, S., Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: The ECCO tool: extraction and composition for clone-and-own. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 665–668. IEEE (2015)
34. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: Proceedings of the European Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE). pp. 295–304, ACM (2007)
35. Kolovos, D.S., Di Ruscio, D., Pierantonio, A., Paige, R.F.: Different models for model matching: an analysis of approaches to support model differencing. In: Proceedings of the Workshop on Comparison and Versioning of Software Models (CVSM), pp. 1–6. IEEE (2009)
36. Kehrer, T., Kelter, U., Pietsch, P., Schmidt, M.: Adaptability of model comparison tools. In: Proceedings of the International Con-

ference on automated software engineering (ASE). pp. 306–309, ACM (2012)
37. Schultheiß, A., Bittner, P.M., Grunske, L., Thüm, T., Kehrer, T.: Scalable N-Way model matching using multi-dimensional search trees. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 1–12. IEEE (2021)
38. Rad, Y.T., Jabbari, R.: 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)
39. Vogel-Heuser, B., Legat, C., Folmer, J., Feldmann, S.: Researching evolution in industrial plant automation: scenarios and documentation of the pick and place unit. Institute of Automation and Information Systems, Technische Universität München, Tech. Rep. (2014)
40. Capozucca, A., Cheng, B., Guelfi, N., Istoan, P.: OO-SPL modelling of the focused case study. In: Proceedings of the International Workshop on Comparing Modeling Approaches (CMA). ACM (2011)
41. Lity, S., Lachmann, R., Lochau, M., Schaefer, I.: Delta-oriented software product line test models - the body comfort system case study. Technische Universität Braunschweig, Tech. Rep. (2012)
42. Martinez, J., Assunção, W.K.G., Ziadi, T.: ESPLA: a catalog of extractive SPL adoption case studies. In: Proceedings of the International Systems and Software Product Line Conference (SPLC), pp. 38–41. ACM (2017)
43. Rubin, J., Czarnecki, K., Chechik, M.: Managing cloned variants: a framework and experience. In: Proceedings of the International Systems and Software Product Line Conference (SPLC). pp. 101–110. ACM (2013)
44. Wenzel, S.: Unique identification of elements in evolving software models. Softw. Syst. Model. (SoSyM) **13**(2), 679–711 (2014)
45. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
46. Jaccard, P.: The distribution of the flora in the alpine zone. New Phytol. **11**(2), 37–50 (1912)
47. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. (TOMS) **3**(3), 209–226 (1977)
48. Schultheiß, A., Bittner, P.M., Boll, A., Grunske, L., Thüm, T., Kehrer, T.: Artifact for RaQuN: A Generic and Scalable N-Way Model Matching Algorithm (2022)
49. Daniel, S., Savarese, F.: libssrckdtree-j. Source Code, available online at https://www.savarese.com/software/libssrckdtree-j/; visited on December 28 (2020)
50. Kuhn, H.W.: The Hungarian method for the assignment problem. Nordic J. Comput. **2**(1–2), 83–97 (1955)
51. Reuling, D., Lochau, M., Kelter, U.: From imprecise n-way model matching to precise n-way model merging. J. Object Technol. (JOT) **18**(2), 1–20 (2019)
52. Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M.: Evolution of software in automated production systems: challenges and research directions. J. Syst. Softw. (JSS) **110**, 54–84 (2015)
53. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Autom. Softw. Eng. **23**(4), 687–733 (2015)
54. Capozucca, A., Cheng, B., Georg, G., Guelfi, N., Istoan, P., Mussbacher, G., Jensen, A., Jézéquel, J.-M., Kienzle, J., Klein J. et al.: requirements definition document for a software product line of car crash management systems. The Repository of Model-Driven Development (ReMoDD) (2011)
55. Kästner, C., Dreiling, A., Ostermann, K.: Variability mining: consistent semiautomatic detection of product-line features. IEEE Trans. Softw. Eng. (TSE) **40**(1), 67–82 (2014)
56. Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A.: Variability extraction and modeling for product variants. Softw. Syst. Model. (SoSyM) **16**(4), 1179–1199 (2017)
57. Krüger, J., Fenske, W., Thüm, T., Aporius, D., Saake, G., Leich, T.: Apo-games: a case study for reverse engineering variability from cloned java variants. In: Proceedings of the International Systems and Software Product Line Conference (SPLC), pp. 251–256, ACM (2018)
58. Schlie, A., Knüppel, A., Seidl, C., Schaefer, I.: Incremental feature model synthesis for clone-and-own software systems in MATLAB/Simulink. ser. SPLC '20. ACM (2020)
59. Schlie, A.: Extractive Product Line Migration for MATLAB/Simulink Software Systems. Ph.D. dissertation, Technische Universität Carolo-Wilhelmina zu Braunschweig (2021)
60. Software plattform embedded systems 'xt' (spes_xt), Website, available online at https://sse.uni-due.de/en/research/projects/spes-2020; visited on February 25th, 2022
61. Collaborative embedded systems (crest), Website, accessed: 2022-02-25
62. Boll, A., Brokhausen, F., Amorim, T., Kehrer, T., Vogelsang, A.: Characteristics, potentials, and limitations of open-source Simulink projects for empirical research. Softw. Syst. Model. (SoSyM) **20**(6), 2111–2130 (2021)
63. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval, vol. 463. ACM, New York (1999)
64. Klatt, B., Küster, M., Krogmann, K.: A graph-based analysis concept to derive a variation point design from product copies. In: Proceedings of the International Workshop on Reverse Variability Engineering (REVE), pp. 1–8 (2013)
65. Schmorleiz, T., Lämmel, R.: 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, pp. 45–48 (2014)
66. Schmorleiz, T.: An annotation-centric approach to similarity management. Master's Thesis, Universität Koblenz-Landau (2015)
67. Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Le Traon, Y.: Bottom-up adoption of software product lines: a generic and extensible approach. In: Proceedings of the International Systems and Software Product Line Conference (SPLC). pp. 101–110. ACM (2015)
68. Ziadi, T., Henard, C., Papadakis, M., Ziane, M., Le Traon, Y.: Towards a language-independent approach for reverse-engineering of software product lines. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 1064–1071, ACM (2014)
69. Selonen, P., Kettunen, M.: Metamodel-based inference of inter-model correspondence. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR). pp. 71–80. IEEE (2007)
70. Kpodjedo, S., Ricca, F., Galinier, P., Antoniol, G., Gueheneuc, Y.-G.: Madmatch: many-to-many approximate diagram matching for design comparison. IEEE Trans. Softw. Eng. (TSE) **39**(8), 1090–1111 (2013)
71. Rubin, J., Chechik, M.: Combining related products into product lines. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 285–300. Springer (2012)
72. Mende, T., Koschke, R., Beckwermert, F.: An evaluation of code similarity identification for the grow-and-prune model. J. Softw. Maint. Evol. (JSME) **21**(2), 143–169 (2009)
73. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: RuleMerger: automatic construction of variability-based model transformation rules. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 122–140. Springer (2016)

74. Fenske, W., Meinicke, J., Schulze, S., Schulze, S., Saake, G.: Variant-preserving refactorings for migrating cloned products to a product line. In: Proceedings of the International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 316–326. IEEE (2017)
75. Reuling, D., Kelter, U., Bürdek, J., Lochau, M.: Automated N-way program merging for facilitating family-based analyses of variant-rich software. Trans. Softw. Eng. Methodol. (TOSEM) **28**(3), 131–1359 (2019)
76. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Automatic variation-point identification in function-block-based models. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE). pp. 23–32. ACM (2010)
77. Wille, D., Babur, Ö., Cleophas, L., Seidl, C., van den Brand, M., Schaefer, I.: Improving custom-tailored variability mining using outlier and cluster detection. Sci. Comput. Program. (SCP) **163**, 62–84 (2018)

Alexander Schultheiß is a doctoral student with Timo Kehrer at Humbold University of Berlin since 2019. Before that, he studied Computer Science at the Fried rich Schiller University Jena. During his studies, he focused on computer vision and software engineering. His main research interests lie in the systematic support of multi-variant software development, with a focus on propagating code changes between cloned variants.

Paul Maximillian Bittner is a PhD student at the Institute of Software Engineering and Programming Languages at University of Ulm. He researches the evolution of feature traceability in variational systems with a focus on creating sustainable clone-and-own development. He got his master's degree from the Institute of Software Engineering and Automotive Informatics at TU Braunschweig in 2020.

Alexander Boll is a doctoral student at University of Bern and is part of the Software Engineering Group since 2022. Before that, he studied computer science at Humboldt-Universität zu Berlin, where he also started his doctoral studies. His research interest is Open Science in the modelling community.

Lars Grunske is currently Professor at the Department of Computer Science from the Humboldt-Universität zu Berlin, Germany. He received his PhD degree in computer science from the University of Potsdam (Hasso-Plattner-Institute for Software Systems Engineering) in 2004. He was Boeing Postdoctoral Research Fellow at the University of Queensland from 2004–2007, a lecturer at the Swinburne University of Technology, Australia from 2008–2011, Junior Professor at the University of Kaiserslautern from 2011–2012, and Professor at the University of Stuttgart from 2012–2015. He has active research interests in the areas modelling and verification of systems and software. His main focus is on automated analysis, mainly probabilistic and timed model checking and model-based dependability evaluation of complex software intensive systems.

Thomas Thüm is a professor for the Construction and Analysis of Secure Software Systems at the University of Ulm since January 2020 and head of the SoftVarE working group. His research interests range from software engineering and formal methods to artificial intelligence and security. In particular, his research focuses on variability and evolution of software systems. From 2015 to 2019, he was a postdoctoral researcher at the TU Braunschweig in Ina Schaefer's institute. He received his Ph.D. in 2015 from the University of Magdeburg under the supervision of Gunter Saake. His Ph.D. thesis received the Dissertation Award 2015 of the University of Magdeburg and his master's thesis the Software Engineering Award 2011 of the Ernst Denert Foundation. He coauthored more than 100 peer-reviewed publications and is known for his contributions to the famous open-source project FeatureIDE. Since 2020, he is an associate editor for ACM Transactions on Software Engineering and Methodology (TOSEM).

Timo Kehrer is a professor at the Institute of Computer Science of the University of Bern (Switzerland), chairing the Software Engineering Research and Teaching Group. Before that, Kehrer was an assistant professor at the Humboldt-Universität zu Berlin (Germany), heading the Model-Driven Software Engineering Group from 2017 to 2021. Kehrer worked as a postdoctoral research fellow in the Dependable Evolvable Pervasive Software Engineering Group at Politecnico di Milano (Italy) from 2015 to 2016, and as a research assistant in the Software Engineering and Database Systems Group of the University of Siegen (Germany) from 2011 to 2015. He has active research interests in various fields of model-based software and systems engineering, with a particular focus on software evolution.