Autor*in

Dimitrios Volikakis

Studiengang

Software Engineering for Embedded Systems, M.Eng.

Masterarbeitstitel

Qualitative Evaluation of N-Way Model Matching Approaches

RPTU
Distance and Independent
Studies Center
DISC

Declaration of Authorship

Ich versichere, dass ich diese Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Place, date

Athens, 31 July 2023

Signature

DIMITRIOS VOLIKAKIS

RHEINLAND-PFÄLZISCHE TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN-LANDAU

Abstract

Software Engineering for Embedded Systems

Master of Engineering

Qualitative Evaluation of N-Way Model Matching Approaches

by Dimitrios Volikakis

In product line engineering tasks, the need for merging models from different product variants emerges as the commonly used clone-and-own approach suffers from high maintenance costs in the long run. By identifying models with a high number of similarities we can merge them to one highly reusable model. This approach will increase the maintainability, and further expandability of the model.

Already many works have been published aiming to solve this problem with different N-way model Matching approaches. However, there is lack of practical evidence that the published theories work as designed in real world cases.

In this work, we will evaluate relevant published approaches and then attempt to integrate the most promising one in the product line analysis framework VARIOUS from Fraunhofer IESE. Next, the implemented approach will be evaluated in comparison to the existing mechanism for model matching that VARIOUS integrates that is called "System Aligner". The main aspects of our evaluation are:

- Accuracy - Can it accurately find the most similar models?
- Performance - How fast is it?
- Scalability - How well does it scale in large amount of input models?
- Configurability - Can it be adapted easily for different systems?

Contents

Contents **iv**

List of Figures **vi**

List of Tables **viii**

1 Introduction **1**
- 1.1 Background and context 1
  - 1.1.1 N-way model matching 2
  - 1.1.2 Impact on software engineering 3
- 1.2 Research questions and objectives 3
- 1.3 Overview of the structure of the thesis 4

2 Related Work **5**
- 2.1 N-way model merging (NwM) 5
- 2.2 Range Queries on N input models (RaQuN) 8
- 2.3 Formalism-based N-way matching algorithm 10
- 2.4 Bioinformatics-based approach 12
- 2.5 Other relevant publications 14

3 Solution Design and Implementation **16**
- 3.1 Selection of N-way model matching approach 16
- 3.2 Deep dive in selected approach: RaQuN 17
- 3.3 VARIOUS Framework 22
  - 3.3.1 Conceptual design of the Framework and its structure 22
  - 3.3.2 The System Aligner component and requirements for RaQuN 24
- 3.4 RaQuN design and implementation for VARIOUS 26
  - 3.4.1 High-level view of RaQuN in VARIOUS 26
  - 3.4.2 RaQuN implementation 27
    - Vectorizer 27
    - Scorer 31

Aligner 34

4 Evaluation **45**
4.1 Evaluation criteria and requirements 45
4.2 Meeting requirements with RaQuN implementation 47
4.3 Meeting requirements with System Aligner 49
4.4 Evaluation results 49

5 Conclusion **52**
5.1 Summary 52
5.2 Future Work 53

Bibliography **55**

List of Figures

1.1 SPL in comparison to a Single System development [1] 2

2.1 NwM Time Complexity 7
2.2 RaQuN vs NwM on large size input models [9] 9
2.3 RaQuN Time Complexity 10
2.4 Formal statements to define similarity metrics for homogeneous models [17] 11
2.5 Comparison operators used in formal statements [17] 12
2.6 Formalism-based algorithm Time Complexity 12
2.7 Bioinformatics-based algorithm Time Complexity 13

3.1 RaQuN algorithm phases [9] 18
3.2 Example: Input Models [9] 19
3.3 Example: Vector Space with two dimensions [9] 19
3.4 Conceptual Design of VARIOUS (DIKW vertically - GQM horizontally) [26] 22
3.5 Logical Structure of VARIOUS and the Analyser Component 23
3.6 VARIOUS Database Class Diagram - Models 24
3.7 VARIOUS Database Class Diagram - Scores 25
3.10 RaQuN Vectorizer - Component Diagram 28
3.11 RaQuN Vectorizer - Python help page 28
3.14 RaQuN Vectorizer - Activity Diagram 30
3.15 RaQuN Vectorizer - Execution 31
3.18 RaQuN Scorer - Python help page 32
3.20 RaQuN Scorer - Activity Diagram 33
3.21 RaQuN Scorer - Execution 34
3.26 RaQuN Aligner - Activity Diagram 36
3.8 VARIOUS Database Class Diagram - Matches 38
3.9 RaQuN implementation - High Level View 39
3.12 RaQuN Vectorizer - Snapshot of `input_models.json` 40
3.13 RaQuN Vectorizer - Class Diagram 40
3.16 RaQuN Vectorizer - Snapshot of `models_vector.json` 41

3.17 RaQuN Scorer - Component Diagram 41
3.19 RaQuN Scorer - Class Diagram 42
3.22 RaQuN Scorer - Snapshot of `models_candidates.json` 42
3.23 RaQuN Aligner - Component Diagram 43
3.24 RaQuN Aligner - Python help page 43
3.25 RaQuN Aligner - Class Diagram 44
3.27 RaQuN Aligner - Execution 44
3.28 RaQuN Aligner - Snapshot of `models_matches.json` 44

4.1 Similarity Score function in evaluation 46
4.2 Accuracy Measure in evaluation 47
4.3 Vectorization with properties as dimensions 48
4.4 Execution Time Results Plot 50
4.5 Matches Quality Results Plot 50

5.1 Weight metric by Rubin and Chechik 53
5.2 Domain agnostic *shouldMatch* formula 53

List of Tables

2.1 NwM algorithm ratings . . . 7
2.2 RaQuN algorithm ratings . . . 10
2.3 Formalism-based algorithm ratings . . . 13

3.1 N-way model matching approaches ratings overview . . . 16

4.1 Evaluation data subjects . . . 46

List of Abbreviations

SPL	Software Product Line
RaQuN	Range Queries on N input models
NwM	N-way model Merging
GQM	Goal Question Metric
DIKW	Data Information Knowledge Wisdom

Chapter 1

Introduction

1.1 Background and context

In today's fast-paced and competitive software development landscape, there is an always growing need for software that is not only functional but also well-designed and highly reusable. This is driven by factors like the increasing complexity of software systems, the need for faster time-to-market, and the desire for cost-effective solutions. In particular, Software Product Line (SPL) engineering, which involves developing a family of related software products using a common set of reusable assets, requires software that is both modular and flexible. The utilization of reusable assets enables organizations to reduce development costs and improve time-to-market by reusing proven, high-quality software components. Figure 1.1 illustrates the payoff of the Product Line Approach comparing to a Single System development.

However, when developing software product lines, developers often clone existing software assets and modify them appropriately for the new product in order to cope with strict deadlines. Following the clone-and-own approach when working with large-scale software systems makes it difficult to manage the variability of these assets across different products. In many cases, multiple versions of the same asset may exist, each with variations in features or functionality.

N-way model matching approaches provide a way to compare and reconcile these different versions of the same asset, by identifying similarities and merging them into a coherent whole. Fully or semi-automating this process can be of big help to ensure that software assets are consistent and coherent across multiple products, reducing development costs and improving the overall quality of the software system.

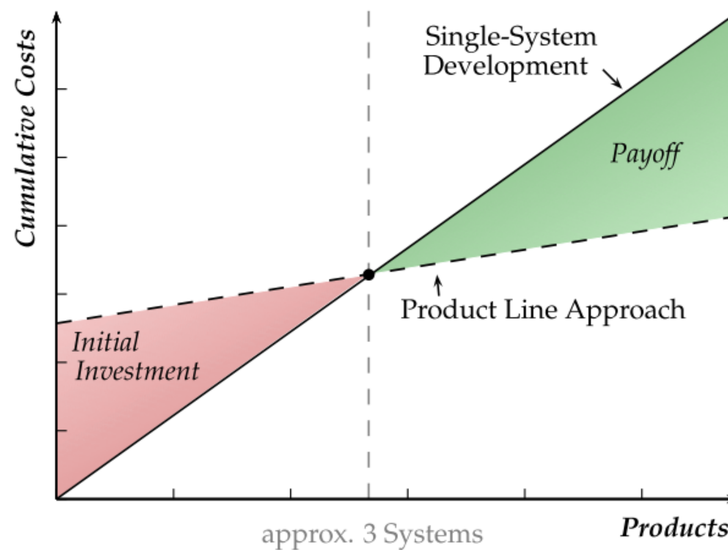

FIGURE 1.1: SPL in comparison to a Single System development [1]

1.1.1 N-way model matching

N-way Model matching has evolved considerably in recent years because of the increasing demand for efficient and effective software engineering practices in the industries [2]. The first works [3]–[5] started appearing in the area of model-driven engineering where the models are used to represent software systems at various levels of abstraction.

Early approaches [6], [7] in model matching were typically pairwise comparisons of models which were lacking in scalability for large software systems. Hence, more recent advances [8]–[11] enabled the comparison of multiple models simultaneously. The term "N-way" refers to the fact that this technique can be used to compare more than two models at a time. These N-way model match techniques leverage algorithms from areas such as graph theory to identify patterns and relationships between different models.

Despite these advances there are still challenges. Efficiency is probably the major one since when the number of models to be compared increases, the computational complexity of the matching process grows rapidly. Additionally, the diversity of modeling languages and tools used in software engineering can be very problematic for model matching. This is because the models may differ in terms of syntax, semantics and structure making it difficult to identify accurately the similarities between the models. Finally, the verification of the matching results in terms of correctness and completeness is another challenge that needs to be addressed for having an effective and robust matching process.

1.1.2 Impact on software engineering

Model matching becomes very useful when it is incorporated in a software product line process for automatically identifying similar software assets that can be potentially reworked before leaving the development phase of a product. By leveraging the similarities between different models, software engineers can develop reusable assets which are used across the product line leading to higher quality software with fewer defects. This results in more efficient and effective solutions that are better aligned with the needs of the organization and its customers.

In addition, N-way model matching fosters collaboration and communication among software engineering teams by providing a shared understanding of the software models being developed for the different products. It helps to reduce misunderstandings and avoid misuse of models that can result in late corrections impacting the product's planned schedule.

The product's compliance checking with regulatory and industry standards can also benefit of model matching by verifying that the software components are developed in accordance with these standards. Predefined standards-based model can be used as golden references to compare against the software components under development. It would be of great help for industries such us healthcare, finance, and aerospace which are very strict in the compliance with standards.

Overall, due to the demand for software systems that are both reusable and adaptable, N-way model matching is likely to continue to be an important area of research and development in the field of software engineering, as it can have a significant positive impact on the quality, reliability and maintainability of the software systems while also improving collaboration and compliance checking.

1.2 Research questions and objectives

The aim of this thesis is to conduct a qualitative evaluation of N-way model matching approaches. To achieve this goal we will start by collecting and reviewing existing literature on N-way model matching approaches in the area of software engineering. Then, based on the findings from the literature review we will select the most promising approach and attempt to implement as part of the product line analysis framework called "VARIOUS" developed by Fraunhofer IESE. The objective of this implementation is to evaluate the feasibility and practicality of an N-way model matching approach in a real-world scenario. The results of this evaluation will provide practical evidence of the applicability and potential benefits of N-way model matching in software engineering. The following research questions are to be addressed:

- ***RQ1***: How well do the existing N-way model matching approaches meet the criteria: Accuracy, Performance, Scalability, Configurability?
- ***RQ2***: What is the implementation and integration process for an N-way model matching approach in a software system?
- ***RQ3***: How does the selected N-way model matching algorithm compare to the existing mechanism of VARIOUS framework?

1.3 Overview of the structure of the thesis

The thesis is structured as follows:

- *Related Work*:
  In this chapter we provide a review of existing N-way model matching approaches that are published in software engineering field. We discuss their strengths and weaknesses aiming to select one of them for implementing and evaluating it.
- *Solution Design and Implementation*:
  This chapter focuses on the design and implementation of the selected approach. We initially justify our choice and then present the VARIOUS platform on which we integrate the N-way model matching technique. We describe the process of integrating it, emphasizing on the architecture and implementation.
- *Evaluation*:
  Here we present the results of our evaluation of the implemented approach. We provide detailed analysis of our findings and compare the applied N-way model matching approach with the pre-existing model matching approach of VARIOUS platform.
- *Conclusion*:
  The final chapter summarizes the results of the study including a discussion of the limitations and directions for future work.

Chapter 2

Related Work

In this chapter we review the N-way model matching approaches that currently exist in academic literature in a chronological order. We aim to find the most promising one for implementing it in a real-world Product line later. The main search criteria is to find model matching approaches utilizing algorithms that process N input models simultaneously and producing model matches as output.

2.1 N-way model merging (NwM)

The first published N-way model matching approach with these characteristics is the N-way model merging (NwM) proposed by Rubin and Chechik in 2013 [8]. This work is motivated by the fact that all previous model matching approaches propose a sequential two-way comparison which may yield suboptimal or even incorrect results because not all input models are considered at the same time. Hence, NwM proposed an N-way model merging approach for the first time in literature.

Firstly, the importance of considering multiple models simultaneously is illustrated with an example where with a pairwise merging approach, a decision made in a certain iteration can impede reaching the desired result in later iterations. It is shown that the pairwise approach shows great sensitivity to the order in which the input models are picked since the matching results are based on limited information rather than the global picture in each iteration.

After having made the need for N-way model merging clear, the study begins with the pairwise model merging attempting to refine the *compare-match-compose* steps of the merge process for N inputs. Tuples are used instead of pairs for the *compare* step where a similarity measure (also referred as weight) is assigned for each one. During the *match* step a validity function decides whether a tuple is eligible to be selected. Finally, the *compose* step combines the elements of each matched tuples. Given that (a) domain-specific information is required for the steps *compare* and *compose* (i.e. which elements are considered similar

and how these elements should be combined) and (b) there are numerous works focusing on these aspects, the study focused on the *match* step of the merging process.

The study continues by showing that the *match* step of the merging process can be reduced to the NP-hard problem of *weighted set packing* [12] whose approximation algorithms [13], [14] prove to be insufficient in terms of scalability for real-life cases of model merging. Therefore, a different approach is explored aiming to provide results polynomial in time for both the number of input models and their size. This is when the NwM algorithm is presented as a novel model merging algorithm that is able to perform well on large scale input models processing them simultaneously.

The main idea of the NwM algorithm is based on picking optimal matches from distinct models and group them incrementally until a maximal set of tuples is produced. More specifically, in the first iteration the elements of all input models are represented individually by single-element tuples. Then, they are assigned weights and get matched using a graph-based match algorithm. The matched tuples are then unified (carrying also the union of their weights) and used as input to the next iteration of the algorithm which finally terminates when no more matches can be made on the input tuples. There can be combinations of tuples that have no similarities and could then be filtered out at the iteration that this is detected. However, because they may participate in a more desired combination at a later iteration, it is preferred to keep them in the inputs by assigning 0 weight on them. Before the algorithm proceeds with the unification of tuples, it first performs a validity check of the new potential unified tuple. If the appended/prepended tuple causes the existing union of chained elements to be invalid, the match with the new tuple is dropped. The last phase of each iteration incorporates an optimization of the constructed chains by checking whether splitting into smaller sub-chains improves the weight of the result.

The NwM algorithm is evaluated by the authors of the study empirically on two example cases by comparing its performance with two subset-based algorithms (these algorithms are based on the pairwise technique only with a larger pool of models). The algorithms defined utilize Greedy [13] algorithm on subsets of size 3 (*Gr3*) and 4 (*Gr4*). The results show that NwM outperforms the subset-based approaches with regard to the weight of the found solution, i.e., the similarity degree of the matched models. In numbers, NwM is able to achieve from 13.5% to 30% improvements compared to Gr3 and Gr4. With respect to the execution time, NwM is sitting in between of Gr3 and Gr4. Comparing with Gr3 which is the fastest of the two, NwM is 67% to 77% slower. Hence, the study concludes that NwM produces results with higher quality and even though its running time is slower, it is still feasible. The complexity of the NwM algorithm is shown in Figure 2.1.

Lastly, in Table 2.1 we attempt to evaluate the NwM algorithm using ratings in the scale of *low*, *medium* and *high* for the important characteristics that we are mainly interested

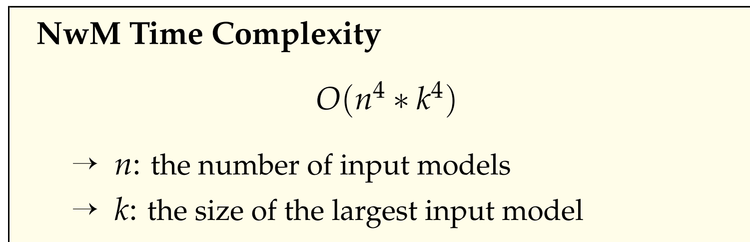

FIGURE 2.1: NwM Time Complexity

(*RQ1*). The ratings are set arbitrarily based on the evaluation data presented by the authors of the algorithm and their conclusion. The evaluation is conducted through a comparison between the proposed approach and the previously predominant method, allowing the authors to assess the quality characteristics:

- *Accuracy*: how accurately can it find the most similar models?
- *Performance*: how much running time it requires?
- *Scalability*: how does it perform when the number of inputs increases?
- *Configurability*: how easy it is to adapt it for different applications?

Characteristic	**Rating**	**Justification**
Accuracy	high	NwM demonstrated a significant improvement in accuracy of up to 30% comparing to subset-based approaches Gr3 and Gr4.
Performance	medium	The execution time of NwM is found to be moderate when compared to subset-based approaches, neither exhibiting the most exceptional nor the poorest performance. It is proven to be slower than Gr3 and faster than Gr4.
Scalability	low	NwM's performance is negatively impacted when used on a large number of input models due to the extensive comparisons required between model elements. In a specific experiment NwM exceeded the 12-hour timeout limit while the subset-based approaches performed in reasonable times.
Configurability	N/A	NwM does not offer configuration capabilities.

TABLE 2.1: NwM algorithm ratings

The N-way model matching approach exhibits performance issues when applied to a large scale of input models, attributed to the vast number of comparisons between model elements. Notably, during one of the experiments involving a set of large-scale models, NwM surpassed the timeout limit of 12 hours, in contrast to the pairwise algorithms.

2.2 Range Queries on N input models (RaQuN)

In 2021, Schultheiß et al. presented RaQuN; a scalable N-way model matching technique using multidimensional search trees [9]. The motivation of this work is to tackle the scalability problems that the NwM algorithm had when applying it to models of realistic size, comprising hundreds of elements (model's discreet entities with name and properties; e.g. a class). The models of this size require a big number of comparisons which leads to performance problems. As an example, given the Runtime Complexity of NwM (Figure 2.1), a set of input data with 100 models, where each model has 25 elements, would result in approximately 3,906,250,000 comparisons.

The key idea behind RaQuN is to represent the elements of all input models as points in a multidimensional vector space using a k-dimentional binary search tree [15] for efficiently finding the nearest neighbors of each element, i.e., the elements which are most similar to a given element. Using only the nearest neighbors for match candidates, reduces the number of required comparisons significantly (more than 90%) promising good performance on big scales without compromising the quality of results.

RaQuN algorithm consists of three phases:

1. *Candidate Initialization*: In the first phase it converts all elements of all input models to numerical vector representations and inserts them into a k-dimentional search tree. The number of dimensions as well as the numerical meaning of each one is up to the user to define.
2. *Candidate Search*: In the second phase, where each model element is mapped to a specific point in the tree's vector space, RaQuN retrieves the nearest neighbors for each of the elements creating sets of match candidate pairs. It only considers as valid neighbor elements the ones that do not belong to the same model as the one under processing.
3. *Matching*: In the last phase RaQuN introduces a measure similar to the weight metric introduced by Rubin and Chechik in NwM algorithm for measuring the similarity of the match candidates. This similarity function is applied on each candidate adding the degree to which the two elements are similar. Finally, the algorithm ends with making tuples of matches by merging the candidate pairs that show sufficient similarity degree according to a down limit that the user of the algorithm has specified.

The main research questions that the authors are called to answer in this work is how RaQuN performs comparing to NwM with respect to running time and quality of results, and subsequently how the algorithm scales with growing model sizes. These questions are answered by using three separate experimental subjects. The first is to use the same example set of models as used in NwM to provide results from a fair comparison, the second is model sets generated from Model-Based Software Product Lines and the last one stems from a software family developed using the clone-and-own approach [16]. Similarly to NwM evaluation, the quality of results is measured with the weight metric and time for the performance. The results showed that RaQuN is significantly faster than NwM on all experimental subjects achieving in parallel the highest weights. With regard to the scalability the evaluation takes place using a fixed number of input model variants and increasing their sizes in number of model elements. Both the running time and the quality of results are measured while increasing the model sizes as it is important that the matching accuracy does not deteriorate on large scales. As shown in the left plot of Figure 2.2 below, RaQuN's running time remains feasible on the large size models in contrast to the NwM. On the right plot we can observe RaQuN producing a slightly higher precision results comparing to NwM. Note that the plots also contain data of Pairwise approaches, but these are considered to be out of scope for this thesis and therefore skipped.

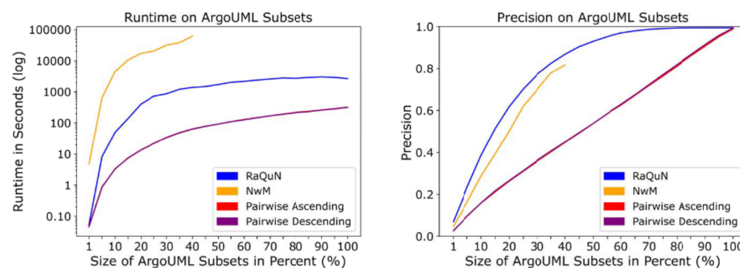

FIGURE 2.2: RaQuN vs NwM on large size input models [9]

The impact of RaQuN's configurability on the algorithm's performance is also assessed by the authors. In fact, the authors measure the running time of RaQuN with different values of the nearest neighbors k' to search as described in the second phase: *Candidate Search* above. The results showed a linear growth of running time with the increase of k'. Also while with the heuristic choice of k'=7 RaQuN retrieves enough candidates for good matches, while still reducing the number of element comparisons by more than 90% for most experimental subjects. The complexity of the RaQuN algorithm is shown in Figure 2.3.

RaQuN Time Complexity

$$O(n^3 * k^3)$$

→ n: the number of input models
→ k: the size of the largest input model

FIGURE 2.3: RaQuN Time Complexity

In Table 2.2 we rate RaQuN algorithm for the characteristics we are interested in:

Characteristic	Rating	Justification
Accuracy	high	RaQuN scored the highest weights in the comparison with NwM and pairwise algorithms across all experimental datasets.
Performance	high	RaQuN outperformed NwM on all input model dataset sizes and proved to be as fast as pairwise algorithms on medium-sized datasets with hundreds of elements.
Scalability	high	RaQuN demonstrated favorable scalability for models of varying sizes, including the largest models in the experimental dataset with over 10000 total elements, and without compromising the quality of matches.
Configurability	medium	RaQuN offers a set of key configuration options, including the definition of vector space dimensions, the number of neighbors to search for each model element in the vector space, as well as the similarity scoring function which determines whether a candidate pair of elements is considered a match or not.

TABLE 2.2: RaQuN algorithm ratings

2.3 Formalism-based N-way matching algorithm

In 2022, Kasai et al. proposed a formalism to facilitate the similarity criteria definition by the users, and subsequently they introduced an N-way matching algorithm based on it [11].

The main motivation of this work is to provide full control to the users over the criteria that determine the similarity between models in an N-way model matching approach.

The N-way model matching formalism is inspired by Rouhi and Zamani specification [17] and its purpose is to specify comparison rules for multiple models which can be either homogeneous or heterogeneous. Homogeneous models are considered the models that refer to the same metamodel, while heterogeneous models refer to different metamodels. With the keyword *module* the name of the comparison module is defined. Each module should have a list of metamodels that will be used for comparison, which are defined with the keyword *inputMetaModels* followed by nested *import* keywords to specify the source URI of the metamodel (e.g. import UML : www.omg.org/UML2/5.0/). Then the comparison rules are defined with keywords *homocomRule* and *hetecomRule* for homogeneous and heterogeneous models respectively. Each rule contains fields for defining its name (*rule*), the related element of the metamodel (*matchHomo*, or *matchHete*) which can be one for homogeneous and many for heterogeneous; and finally a block with keyword *compareModels* that specifies the details of model elements in the comparison process. This block contains a list of formal statements (*homoStatement*, or *heteStatement*) that define the similarity metrics according to the input metamodels (Figure 2.4). The comparison operators (*compOp* and *comparisonOp*) define logical operators with textual or mathematical notations (Figure 2.5). The definition of the reference element of the metamodel is specified with the keyword *modelElement* which contains the element type and properties.

$$
\begin{aligned}
&homoStatement \quad ::= \\
&\quad [[`\neg'],\ compOp,\ `(',\ modelElement,\ `)']^{+}\ | \\
&\quad [`isValid',\ `(',\ modelElement,\ comparisonOp, \\
&\quad [modelElement\ |\ name : String],\ `)']^{+}\ | \\
&\quad `(',\ [[`\vee'],\ homoStatement]^{*}\ | \\
&\quad [[`\wedge'],\ homoStatement]^{*},\ `)' \qquad (8)
\end{aligned}
$$

FIGURE 2.4: Formal statements to define similarity metrics for homogeneous models [17]

The proposed N-way model matching algorithm that is based on the formalism, receives a set of input models and produces a list of matched model elements. It iterates through all input model elements assigning each element in a group (also mentioned as *chain*) with similar elements. Each element has to satisfy the formalism-based similarity metrics defined by the user. Finally, a list of groups is made where each group contains model elements that satisfy the user's similarity rules and conform to each other. As the authors explain, the formalism narrows down the comparison search area, because the algorithm

$$compOp ::= \text{`}isEqual\text{'} \mid \text{`}isTrue\text{'} \mid \text{`}isFalse\text{'} \mid \text{`}isEmpty\text{'} \mid \text{`}isContain\text{'} \mid \text{`}isEquivalent\text{'} \quad (9)$$

$$comparisonOp ::= \text{`} < \text{'} \mid \text{`} > \text{'} \mid \text{`} \leq \text{'} \mid \text{`} \geq \text{'} \mid \text{`} = \text{'} \mid \text{`} <> \text{'} \quad (12)$$

FIGURE 2.5: Comparison operators used in formal statements [17]

only considers the elements that belong to the formalism, while the elements that are not in the domain of the comparison are ignored. This effectively reduces the number of comparing operations. The time complexity of the Formalism-based algorithm is presented in Figure 2.6.

Formalism-based algorithm Time Complexity

$$O(n * m^2)$$

→ n: the number of input models

→ k: the size of the largest input model

FIGURE 2.6: Formalism-based algorithm Time Complexity

In terms of evaluation, the authors of this work did not present any data. Therefore, it is hard to assess Accuracy, Performance and Scalability characteristics for our review. However, the Configurability is considered as the main asset of this approach as the proposed approach gives the full control of the comparison criteria to the user and hence it is easy to adjust for the needs of the target software system.

2.4 Bioinformatics-based approach

The VARIOUS framework by IESE, where we plan to implement and integrate the chosen approach after the thorough review in this chapter, has already a model matching technique incorporated against which we will evaluate the chosen approach. This technique is based on the Thesis work of Vasil Tenev for his Bachelor's degree [10] and the software component that implements it in VARIOUS framework is called *System Aligner*.

Characteristic	Rating	Justification
Accuracy	N/A	No evaluation data available.
Performance	N/A	No evaluation data available.
Scalability	N/A	No evaluation data available.
Configurability	high	With this approach the users can define their own requirements for the comparison process of the algorithm at model element level which makes it highly configurable.

TABLE 2.3: Formalism-based algorithm ratings

The work [10], [18] presents an approach for the simultaneous analysis of software variants across multiple systems. The algorithms employed in this work are inspired by bioinformatics, specifically DNA sequence alignment. Similar to how nucleic acid sequences describe the anatomical and functional characteristics of living organisms, source code describes the structural features and behavior of software systems.

The study begins with constructing a data model of systems by defining software systems as directed colored multigraphs. The existing theory of DNA alignment is then extended to the new data model. Then the author defines formally the problem of optimal software systems alignment, and explores the mathematical bounds. It is discovered that the problem is exponential, rendering it impractical for real world cases due to the high computational cost. Therefore, a heuristic solution with polynomial-time complexity is sought for the problem. An 8-approximation method for pairwise alignment of software systems is devised and combined with an iterative variant of the Center Star method by Gusfield [19]. The resulted algorithm creates a method for variant analysis with time complexity shown in Figure 2.7.

Bioinformatics-based algorithm Time Complexity

$$O(n^3 * k^2 * log(k))$$

→ n: the number of input models

→ k: the size of the largest input model

FIGURE 2.7: Bioinformatics-based algorithm Time Complexity

The bioinformatics-based approach will not be part of our evaluation in this chapter's

review of existing N-way model matching approaches, as it is the approach that VARIOUS framework has grounded in its model matching function with the System Aligner component. The results of our selected approach will be compared with the ones of System Aligner.

2.5 Other relevant publications

Apart from the three aforementioned N-way model matching algorithms there are many publications about model merge refactoring approaches which in essence focus on migrating a set of variants into an integrated software product line. These works do not fully relate to the scope of this thesis as they present heuristic approaches rather than algorithms that can be implemented. However, it is worth citing them as part of this chapter.

- *From Imprecise N-Way Model Matching to Precise N-Way Model Merging.* [20]

  Generally, the N-way model merging techniques are based on three operators (1) compare, (2) match and (3) merge. The last step requires as input the results of the N-way model matching which incorporates the compare and match steps. This work focuses on the merge step proposing a methodology which incrementally applies as a post-processing step model-refactoring operators, to identify and unify further similarities among (initially) unmatched model elements.
- *Automated N-way Program Merging for Facilitating Family-based Analyses of Variant-rich Software.* [21]

  In this work, a methodology named SiMPOSE is proposed, which incorporates automatic generation of superimpositions of N given program versions and/or variants (in C language) to facilitate family-based analyses of variant-rich software. It is based on an N-way model merging technique operating at the level of control-flow automata (CFA) representations of C programs.
- *Identifying and Visualising Commonality and Variability in Model Variants.* [22]

  A systematic comparison method is presented in this work, namely MoVaC (Model Variants Comparison), which focuses on the comparison of a set of model variants to identify commonalities and variabilities in the form of features. The authors based their second work [23] on MoVaC to build a Model-based Software Product Line by re-engineering the model variants.
- *SYS2VEC: System-to-Vector Latent Space Mappings* [24], [25]

  In this paper, the SYS2VEC approach is presented, which focuses on mapping product line variants into a latent vector space by means of machine learning techniques. It

exploits the stochastic nature of the machine learning models to decrease the time taken for pairwise similarity comparison using row vectors in the finite-dimensional vector space.

Chapter 3

Solution Design and Implementation

This chapter centers on the design, implementation, and integration of an N-way model matching approach as part of the product line analysis framework VARIOUS. Initially, we describe the selection process of the most promising approach, based on the review of existing studies presented in Chapter 2, and then provide an in-depth analysis of the algorithm used in the selected approach. Finally, we discuss the VARIOUS framework, which serves as the basis for the design and implementation of the N-way model matching algorithm.

3.1 Selection of N-way model matching approach

In Chapter 2 we did a review of all existing N-way model matching approaches. We defined the main criteria for our evaluation and rated all approaches accordingly. In Table 3.1 we present an overview with all the ratings of the reviewed approaches. Apparently, the Bioinformatics-based approach discussed in 2.4 is excluded from the selection process as it is the one that currently exists in VARIOUS and will act as the counter approach against which we will put the selected one for comparison.

Characteristic	**NwM**	**RaQuN**	**Formalism-based**
Accuracy	high	high	N/A
Performance	medium	high	N/A
Scalability	low	high	N/A
Configurability	N/A	medium	high

TABLE 3.1: N-way model matching approaches ratings overview

After thorough consideration, it has been concluded that the Formalism-based approach, discussed in 2.3 should not be included in further analysis. Although it has been shown to have better time complexity when compared to the other candidates, it lacks sufficient evaluation data to substantiate its actual performance, which is essential for this Thesis in determining the effectiveness of the approach.

This leaves us with the remaining approaches namely NwM and RaQuN that were reviewed in 2.1 and 2.2 respectively. The former was the first N-way model matching approach that paved the way for the development of RaQuN, which addresses the main scalability issue of NwM. RaQuN provides a high level of *Configurability* to the user, allowing the adaptation of the algorithm to any target software system, and the tuning of the model matches *Accuracy* at the cost of a linear runtime increase. According to the evaluation data, RaQuN outperforms NwM in terms of *Performance* on all test datasets. It is evident that the *Scalability* and *Configurability* of RaQuN make it an appealing approach for model matching in modern software product lines that typically involve hundreds of model elements. Therefore, we consider RaQuN as a suitable candidate for further analysis and integration into the VARIOUS framework.

3.2 Deep dive in selected approach: RaQuN

As mentioned in the preceding chapter's analysis of the RaQuN approach (2.2), the algorithm comprises three phases. The objective of the first two phases, namely *Candidate Initialization* and *Candidate Search*, is to minimize the number of comparisons needed during the last phase: *Matching*. In Figure 3.1 the algorithm is shown with the three phases highlighted in colors.

In *Candidate Initialization* phase, RaQuN collects all input model elements and constructs a multidimensional binary search tree (or k-d tree, where k is the dimensionality of the search space) [15] consisting of the model elements' vector representations as nodes in the tree. The vectorization of the input model elements is defined by the user of the algorithm. In essence, the vectors (or dimensions of the vector space) can be any characteristics of the model elements that the user wishes to base the comparison on in the next phase. A simple example given by the authors uses as inputs three models, each one having a number of model elements and each element containing a number of properties as shown in Figure 3.2. In the example, a two-dimensional vectorization is set, with the first dimension being the average length of all elements' property names, and the second the number of properties of an element. The vectorization is illustrated in Figure 3.3 where the input model elements are mapped in the vector space. The selection of the dimensions in the vector space is very

Algorithm 1 RaQuN

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

FIGURE 3.1: RaQuN algorithm phases [9]

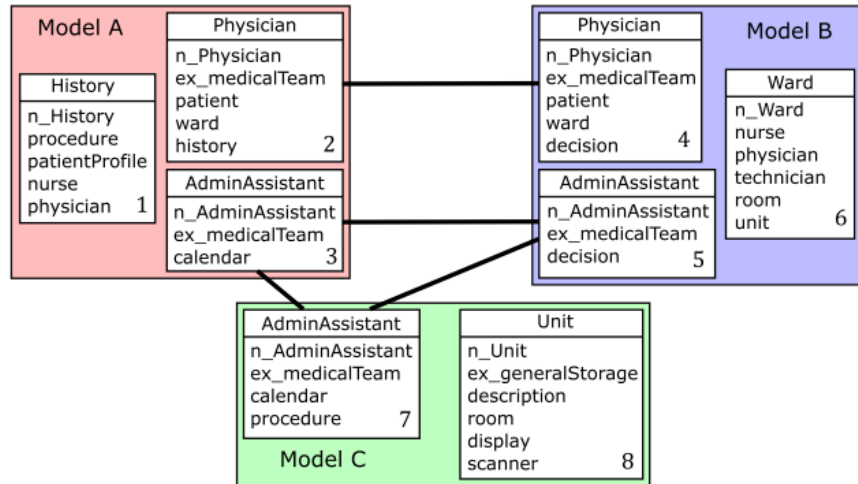

FIGURE 3.2: Example: Input Models [9]

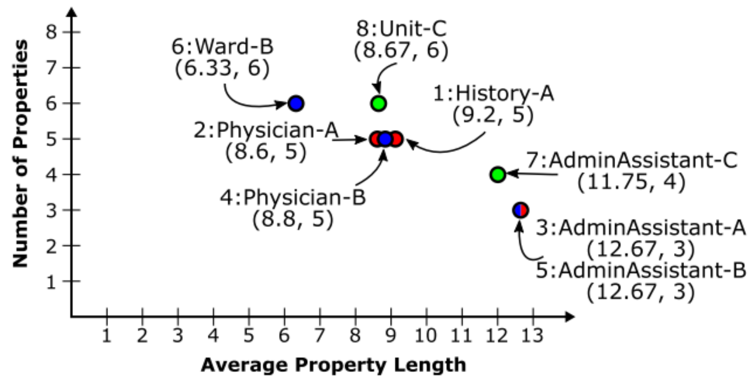

FIGURE 3.3: Example: Vector Space with two dimensions [9]

important for the next phase where each element will be examined individually for finding its closest neighbors to conduct a set of match candidate pairs.

In the *Candidate Search* phase, RaQuN uses the k-d tree search function [15] with a suitable distance metric, such as the Euclidean distance, to retrieve a specified number of nearest neighbors as defined by the user. For each element and its corresponding neighbors, RaQuN creates match candidate pairs, forming a set of pairs to pass on to the subsequent phase. To reduce the number of comparisons, the algorithm considers a match candidate pair as valid only if the two elements of the pair belong to different models. It is worth noting that the k-d tree search function also includes the self-element as a neighbor in the results. Having a closer look at Figure 3.3 we can see two elements, namely `3:AdminAssistant-A` and `5:AdminAssistant-B`, sharing the same point in the vector space. In such cases, where multiple elements possess identical vector representation, RaQuN automatically expands the number of neighboring elements to ensure that all sufficiently close elements are taken into consideration for the comparison. In the given example, three neighbors were selected, resulting in the final set of match candidate pairs:

$$P = (\{1,4\}, \{2,4\}, \{3,5\}, \{3,7\}, \{5,7\}, \{5,1\}, \{6,2\}, \{6,8\}, \{7,1\}, \{8,2\}, \{8,4\})$$

The last phase is *Matching* where RaQuN compares the elements of each match candidate pair based on a user-defined similarity function. The objective of this function is to assess the degree of similarity between the two elements by producing a similarity score. After applying the similarity function, a new reduced set of pairs is created sorted by their score. The pairs with zero similarity score are excluded. In the provided example, a simple similarity function is employed which calculates the ratio of shared properties to all properties. For instance, in the case of the match candidate pair $\{3,7\}$, the function assigns a score of $\frac{3}{4}$, as the model elements 3 and 7 share three properties (`n_AdminAssistant`, `ex_medicalTeam`, `calendar`) out of the four properties present in their combined set (`n_AdminAssistant`, `ex_medicalTeam`, `calendar`, `procedure`). The set created after applying the similarity function in our example is:

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

Next, RaQuN creates the final set of matches which is the output of the algorithm. It initializes the final set of matches by adding all individual input model elements:

$$T = (\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\})$$

This acts as the ground for building the matched elements based on the similarity score

calculated earlier. RaQuN then retrieves each match candidate pair from $\hat{P}$ and decides whether the two elements will be merged or not into a match tuple depending on a lowest score criteria that the user defines. For the sake of simplicity, the threshold given in the provided example is $\frac{1}{2}$, meaning that any candidate pair with score greater or equal to $\frac{1}{2}$ should be merged. Starting with the match candidate pair having the higher score in our example, we see that the score $\frac{3}{4}$ satisfies the lowest score criteria, hence RaQuN merges tuples $\{3\}$ and $\{7\}$:

$$T = (\{1\}, \{2\}, \{\mathbf{3}, \mathbf{7}\}, \{4\}, \{5\}, \{6\}, \{8\})$$

Subsequently, the pairs $\{2, 4\}$, and $\{3, 5\}$ also satisfy the score criteria, hence the tuples including their elements are merged:

$$T = (\{1\}, \{\mathbf{2}, \mathbf{4}\}, \{\mathbf{3}, \mathbf{5}, 7\}, \{6\}, \{8\})$$

Lastly, the pairs $\{5, 7\}$, $\{7, 1\}$ and $\{6, 8\}$ do not pass the lowest score threshold hence the unmerged elements remain in single tuples in the final result:

$$\boxed{T = (\{1\}, \{2, 4\}, \{3, 5, 7\}, \{6\}, \{8\})}$$

Upon examining the application of RaQuN in the provided example, it becomes evident that the algorimth exhibits a notable degree of configurability, confirming the favorable assessment from the previous chapter's evaluation. The points of variation offered by the algorithm are outlined below.

In the Candidate Initialization phase, users have the option to define the dimensions of the vector space, facilitating the clustering of similar elements within the same region of the vector space. Increasing the number of dimensions increases the likelihood of clustering similar elements, however it is important to note that a higher number of dimensions can adversely affect performance.

Moving to the Candidate Search phase, users have the flexibility to specify the number of neighboring elements to be retrieved for each element. This parameter directly influences the number of candidate pairs considered during the matching phase. Additionally, users can define the distance metric to be used in determining the distance between vector representations of two elements within the vector space.

In the Matching phase, users can define the similarity function used to evaluate the quality of a match. Furthermore, they can specify the matching criteria considered by RaQuN to determine the sufficiency of a score generated by the similarity function.

By offering these configurable options, RaQuN empowers users to adapt the algorithm's behavior to their specific needs, enhancing its versatility and applicability in various contexts.

3.3 VARIOUS Framework

When managing a software product line it is important to take into account the financial implications of the incorporated features. The economics of a feature should be justified by its value to the potential customers in relation to the cost that it takes for maintenance and further expansion. To make this assessment, multiple sources are required to extract the necessary data for processing and analysis. Apart from the thorough understanding of the product line itself, the customer needs, and the market trends, it is vital to employ expertise in data analysis and modelling. This can be time-consuming and labour-intensive process, hence there is a need for an effortless decision support to the product line analyser.

3.3.1 Conceptual design of the Framework and its structure

VARIOUS is an approach developed by IESE [26] that presents a method for analysing software product lines (SPLs) in order to identify and address inefficiencies resulting from excessive variability. It combines the Goal-Question-Metric (GQM) approach [27] with the Data-Information-Question-Wisdom hierarchy (DIKW) [28] to facilitate informed decision-making. The GQM approach ensures alignment between a company's business and engineering strategies, while the DIKW framework enables thorough analysis of data to extract valuable knowledge and gain a holistic understanding of the costs and customer value associated with each feature. By leveraging these two approaches, the VARIOUS method constructs a multimodel knowledge graph that serves as a framework for conducting data analyses and addressing complex questions related to scoping, maintaining, and optimizing SPLs.

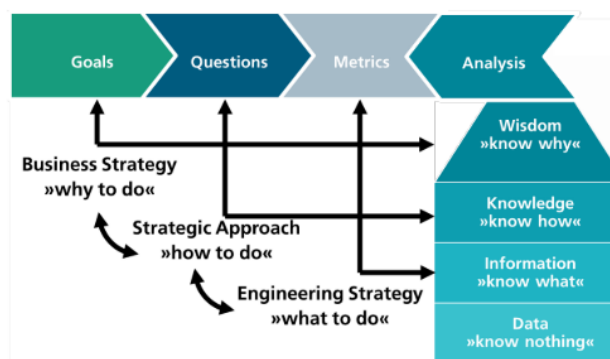

FIGURE 3.4: Conceptual Design of VARIOUS (DIKW vertically - GQM horizontally) [26]

The knowledge graph is constructed based on the existing data models of the SPL. The construction of the graph involves analyzing all artifacts of the SPL building collections to determine their similarity. This analysis facilitates the transition from "Data" to "Information" in the DIKW hierarchy. VARIOUS employes a comprehensive database of data models and incorporates the *System Aligner* component to perform the analysis of these data models. The System Aligner utilizes a similarity scoring function to establish similarity metrics, which are then used to generate tuples of model matches, resulting in a knowledge model that captures the variability of the product line. "Knowledge" is the next phase of DIKW, and its accuracy is important for the decision support VARIOUS aims to provide to the user.

In this thesis, our focus is on the aforementioned phases of VARIOUS, specifically the ones implemented by the System Aligner. As discussed in Section 2.4, the System Aligner implements the bioinformatics-based N-way model matching approach originally presented by Vasil Tenev in his studies [10], [18]. To address the second research question (*RQ2*) of this thesis, we utilize the VARIOUS framework as the foundation for implementing and integrating RaQuN approach, which would act as an alternative to the System Aligner component. The ultimate goal is to evaluate the performance and efficiency of these two approaches by using the model data in the database of VARIOUS.

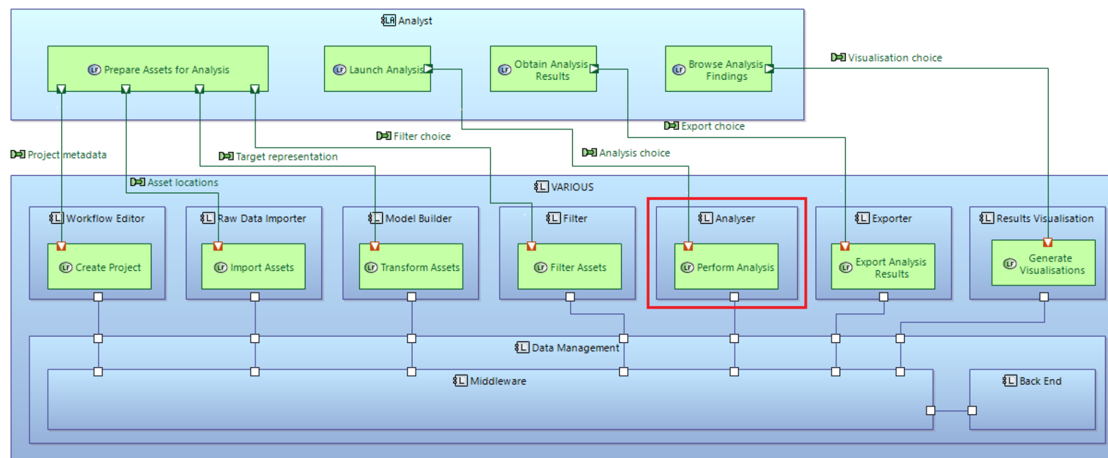

FIGURE 3.5: Logical Structure of VARIOUS and the Analyser Component

VARIOUS framework consists of a core block called *Data Management* and several other blocks that offer the functions to the outside utilizing the core block. Data Management encompasses the *Back End* and the *Middleware* sub-blocks. The Back End contains the database housing the models while the Middleware serves as an interface between the Back End and the blocks consuming the model data. Figure 3.5 provides a visual representation

of the logical structure of the VARIOUS framework, illustrating the operations available to users. The block diagram depicts the Analyst as the primary actor, with the green blocks representing the operations offered by the VARIOUS framework for the Analyst to utilize. Of particular interest to us is the highlighted block called *Analyser* where the System Aligner component resides. The Analyser offers the function *Perform Analysis* which conducts the analysis of the model data retrieved from the Back End database. In the next chapters we discuss how a new component that implements the RaQuN N-way model matching approach can be integrated in the VARIOUS Analyser.

3.3.2 The System Aligner component and requirements for RaQuN

The database of VARIOUS residing in the Back End block of Data Management, is organized in *Collections* of *Documents* and *Edges*. The Document Collections contain the models represented as assets, while the Edge Collections contain the information of how assets are associated to each other. As depicted in the Class Diagram of Figure 3.6, the model assets have a field to be uniquely idenfiable: `_id`, and the edges link two assets with `_from` and `_to` predicates containing the field `_type` to indicate the type of association between the assets (e.g. `_type = "has"`).

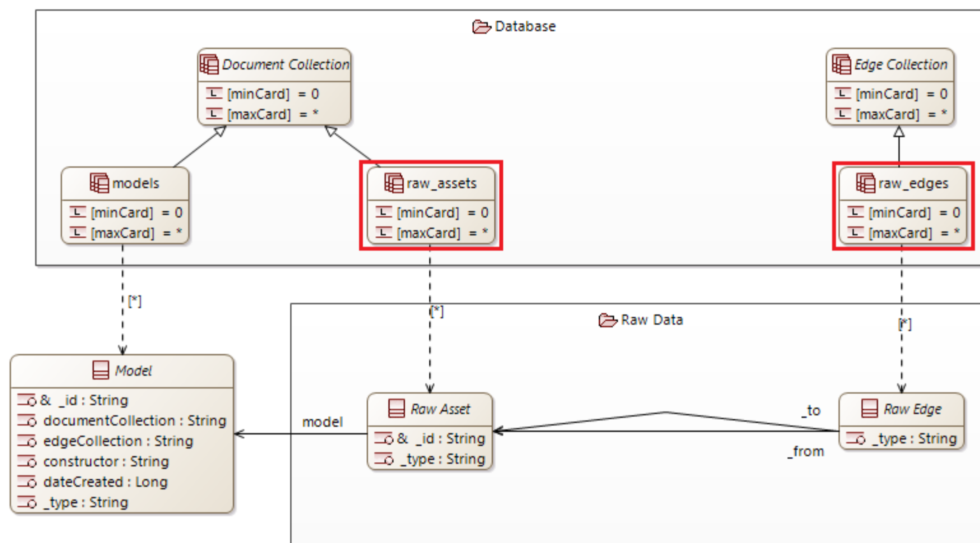

FIGURE 3.6: VARIOUS Database Class Diagram - Models

The System Aligner component interacts with the database for two main purposes. The first and foremost is to analyse the model data to measure their similarities. This task is

accomplished by the *Scoring* module which incorporates a *Scoring Function* to calculate the similarity score of models deemed as match candidates. As shown in Figure 3.7, the scores are stored in the database as edges within the Edge Collection. These edges are identified with `_type = "scoring_edge"` and include a second field called `score` that holds the actual similarity score as an integer, reflecting the degree of similarity between the associated model assets. The second purpose of System Aligner component's interaction with the

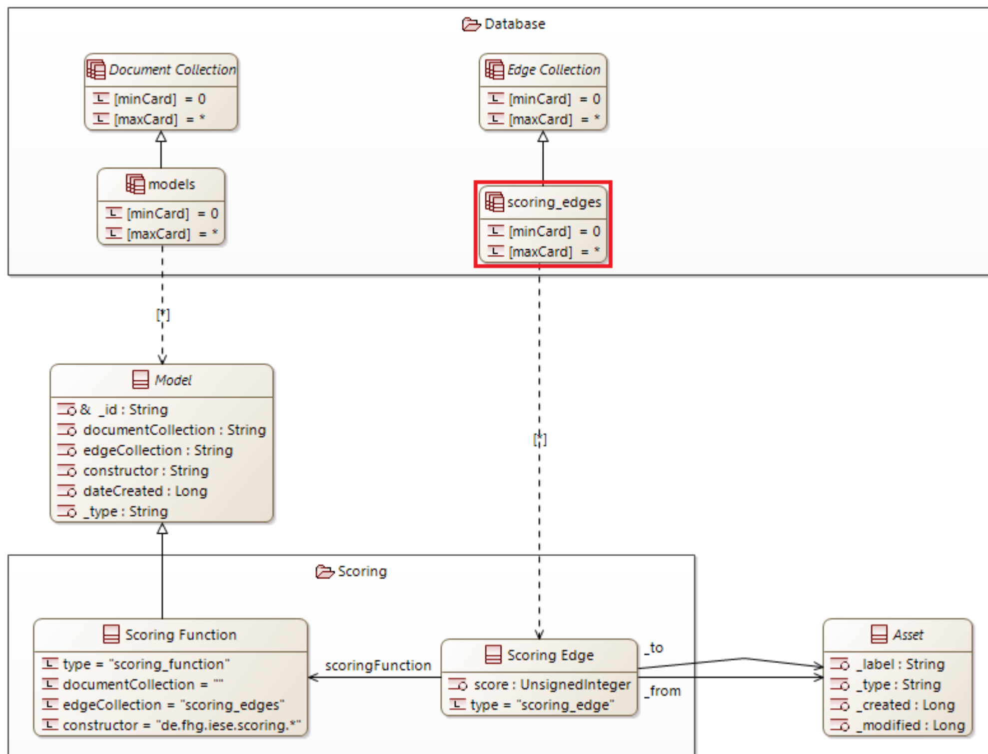

FIGURE 3.7: VARIOUS Database Class Diagram - Scores

database is to reconcile the model match candidates outputted by the Scoring module, into a set of model matches. The *Aligner* module analyses the scores stored in the database and determines which model assets qualify as valid matches based on their similarity score. The resulting model *match_tuples* are stored in the database within the Document Collection and are associated with edges of `_type = "match_tuple_edge"` from the Edge Collection (Figure 3.8).

After examining the role of System Aligner in VARIOUS, and understanding its interaction with the database, we can proceed with planning the integration of the RaQuN

approach. The implementation of RaQuN should be carefully designed to seamlessly replace the System Aligner, without impacting any other components. This entails ensuring that the interfaces of System Aligner remain intact, and RaQuN is implemented in such way that the overall behaviour of the Analyser block is preserved.

It is essential to ensure that RaQuN serves the same purposes as the System Aligner, the model similarity scoring and the model match alignment. Both scoring and match alignment outputs generated by RaQuN should adhere to a format that is compatible with the database. VARIOUS utilizes the JSON format for its database interface, which follows a predefined layout. The same format is used for the input models exported from the database, hence RaQuN must be capable of handling these data, enabling smooth compatibility.

3.4 RaQuN design and implementation for VARIOUS

3.4.1 High-level view of RaQuN in VARIOUS

In order to incorporate the RaQuN approach into the VARIOUS Framework, our initial step is to map the phases of the algorithm onto the modules Scoring and Aligner of the Analyser block in VARIOUS. As mentioned in Section 3.2, the RaQuN algorithm comprises three phases: *Candidate Initialization*, *Candidate Search* and *Matching* (refer to Figure 3.1). To begin this mapping process, we can consider introducing the two modules *RaQuN Scorer* and *RaQuN Aligner* similarly to the existing ones in Analyser. RaQuN Scorer would primarily handle the first two phases of the algorithm, plus the initial part of the last phase where the similarity function is applied (Figure 3.1 line 18). The output of RaQuN Scorer would be the similarity score of the input models, while the RaQuN Aligner module would handle the remaining part of the algorithm which involves the model match candidates alignment, generating the final set of model matches tuples.

The RaQuN algorithm relies heavily on transforming the input model data into points within a k-dimentional vector space, aiming to leverage the efficient search capabilities of the k-dimensional tree structure [15]. Given the significance of this task and the need to maintain modularity in the implementation, it is essential to introduce a dedicated software module for this purpose named *RaQun_Vectorizer*.

Figure 3.9 illustrates the mapping of the RaQuN algorithm's phases onto the three software modules we have discussed. Each module is defined with its respective input and output data, along with the main activities performed, referencing the corresponding lines of the RaQuN algorithm (Figure 3.1). Additionally, the integration with the VARIOUS database through JSON formatted files is also depicted.

In the subsequent paragraphs we discuss the implementation aspects associated with each of the software modules.

3.4.2 RaQuN implementation

The choice of the Python programming language was made for the actual development of the software modules. Python is renowned for its simplicity and readability due to its expressive syntax that allows for writing concise and understandable code. Furthermore, Python comes with built-in integration with JSON that allows efficient handling of data in this format. Tasks like file parsing, data serialization and deserialization, and file writing are extremely simplified. This significantly reduces the effort required for interfacing the database of the VARIOUS framework.

The RaQuN algorithm offers many variation points that allow for tailored implementation according to the specific requirements of the user. By carefully considering and making appropriate decisions on these variation points, the accuracy of the resulting model matches can be significantly enhanced. In the next paragraphs we see all these variation points exposed in each software module. For the sake of demonstration we have based our implementation decisions on the example outlined in the publication [9]. Our objective is to reproduce the result of the exact same model matches as in the example, using the same set of provided input model data. While some variation points allow for being set dynamically, others require additional static adjustments. To ensure flexibility and adaptability, our implementation is focused on code configurability; so that users can easily alter the variation points to their own needs.

Following the RaQuN work, the implementation of the algorithm in this thesis considers the model data representation to follow the *element-property approach* [8]. A model M of size m is a set of $elements\{e1, ..., e^m\}$. Each model element $e \in M$, comprises a set of $properties$. For this running example, the model $elements$ are UML classes, and the $properties$ are the attributes of these classes.

Vectorizer

The RaQuN Vectorizer serves as the entry point in the implementation of RaQuN within the VARIOUS framework. Its scope is to convert the input model data, exported from the database, into an appropriate format as required by the RaQuN algorithm. This involves transforming each element of the models into a corresponding point in the vector space, thus creating a vector space representation. The dimensions (or vectors) in this space are user-defined. In our example-based implementation the vectors chosen are two: the average length of the properties of each model element, and the number of properties of

each model element. Additionally, the user shall provide the input model data from the VARIOUS database in JSON format, and optionally the directory where the resulted file will be stored. The depicted Figure 3.10 provides an overview of the overall inputs and outputs of RaQuN Vectorizer and in Figure 3.11 the Python command-line *help* page is shown.

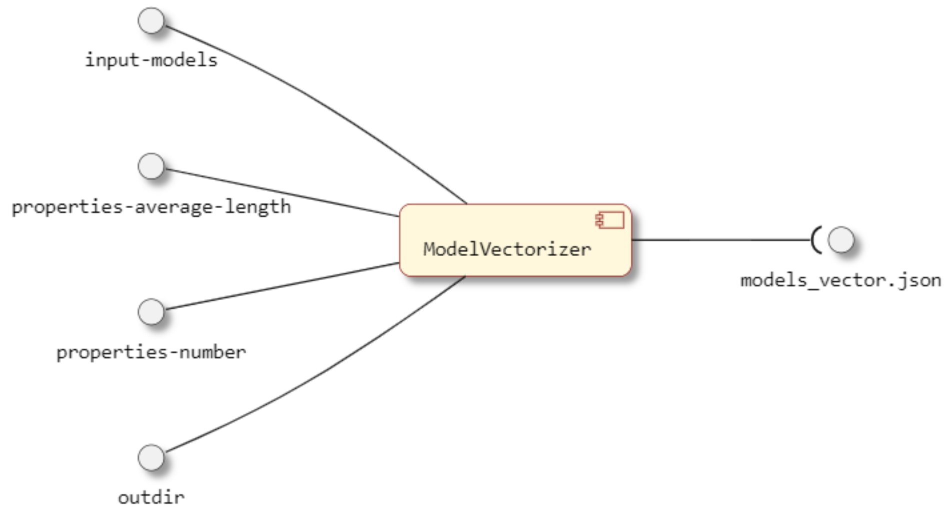

FIGURE 3.10: RaQuN Vectorizer - Component Diagram

```
> python .\raqun_vectorizer.py -h
        usage: raqun_vectorizer.py [-h] -inmodels _INMODELS [-propavglen] [-propnum] [-o _OUTDIR]

        The RaQuN Vectorizer component puts the input model elements into a vector space with
        dimensions defined by the user.

        options:
          -h, --help            show this help message and exit
          -inmodels _INMODELS, --input-models _INMODELS
                                The absolute path to the .json file with the input model data
          -propavglen, --properties-average-length
                                Add dimension with the average length of the properties of each
                                model element
          -propnum, --properties-number
                                Add dimension with the number of properties of each model element
          -o _OUTDIR, --outdir _OUTDIR
                                The directory where the generated file will be stored.
                                If not defined, the current will be used.
```

FIGURE 3.11: RaQuN Vectorizer - Python help page

The *input-models* refers to the JSON formatted model data from VARIOUS database. It is a mandatory argument so if not provided an error will be returned. Figure 3.12 depicts a snapshot of the contents of the `input_models.json` file. The data is structured as an array of models where each model entry contains arrays of `components` and `connections`. The components represent the model elements and properties as entities following the *element-property approach* mentioned earlier. Each component has a member `type` to specify its type as `uml-element` or `uml-element-property` and the members `_id` and `name` to be uniquely identifiable. The relationship between model elements and properties is specified with the array of connections. Each connection has the members `_from` and `_to` whose values are component `_id`(s) and the member `type` that indicates *has* relationship. It is worth noting here that when RaQuN Vectorizer parses the input models, it determines as valid relationships only the ones where the `_id` of `_from` member corresponds to a component of `type` `uml-element` and the `_id` of `_to` member corresponds to a component of `type` `uml-element-property`.

RaQuN Vectorizer is implemented in Python language as a *class* which offers certain APIs. These APIs are then used by `__main__` block of the Python script in combination with the arguments passed by the user on run-time. The *Class Diagram* in Figure 3.13 illustrates the private members and public APIs of the class named `ModelVectorizer`. The class constructor requires the `input_models.json` file to be provided in order to parse the data and create an internal data structure (`models{}`) out of it; easy to use for the subsequent actions. Having created an object of the class, the API `add_dimension()` is used to specify which dimensions will be utilized during vectorization. The supported dimensions (properties average length and number of properties) are pre-defined in the implementation and are offered as arguments to the user on run-time to select. With the API `vectorize()` the actual vectorization takes place that converts the model elements into points of a vector space. The internal data structure is traversed calculating the vector values of each model element for the selected dimensions. Each dimension is bound to an internal function that calculates the vector value of the given model element.

Figure 3.14 summarizes the aforementioned logic of RaQuN Vectorizer showing the flow of actions on run-time with an *Activity Diagram*. Firstly, the provided arguments are used to create an object of the class and add dimensions to it. Then the actual vectorization happens generating a file with the results at the output directory which is optionally set by the user. A successful execution of the RaQuN Vectorizer Python script is depicted in Figure 3.15.

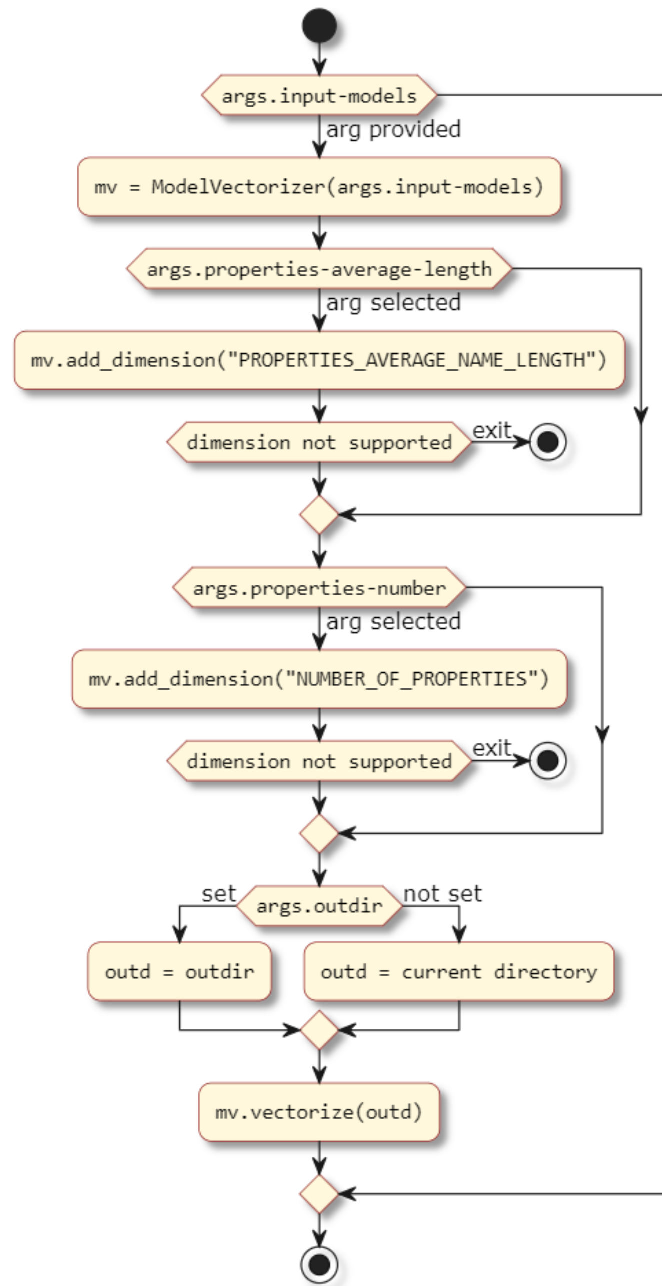

FIGURE 3.14: RaQuN Vectorizer - Activity Diagram

```
> python .\raqun_vectorizer.py -inmodels .\input_models.json -propavglen -propnum -o .
        The vector json file was generated successfully:
        .\models_vector.json
```

FIGURE 3.15: RaQuN Vectorizer - Execution

The data in the generated file `models_vector.json` (Figure 3.16) is structured as an array of points. Each point contains the actual vector values as well as some meta-data of the corresponding model element which are needed by the consumers of this file. The meta-data consist of the element's id and name, the model that the element belongs to, and its properties. Moreover, for better traceability it contains a timestamp, and the inputs (input models data and dimensions) that resulted to this file.

Scorer

The next step in the implementation is the integration of RaQuN Scorer, which takes as input the output generated by RaQuN Vectorizer. The objective of RaQuN Scorer is to identify potential match candidates for each vectorized model element. This is achieved by utilzing the k-d tree data structure and employing the algorithm determining the k nearest neighbors. For each pair of match candidates, RaQuN Scorer assigns a similarity score in the form of a percentange. The user is prompted to provide two inputs (apart from the *models-vector* outputted by RaQuN Vectorizer): the value of `k`, representing the number of nearest neighbors to search for each model element, and the scoring function used to calculate the similarity score. The former is a dynamically supplied integer parameter, while the latter requires a static implementation for each supported scoring function. The user can select the desired scoring function by specifying the corresponding argument in the program. In the current implementation, a simplistic scoring function has been implemented and made available to the user. This function calculates the similarity score by computing the ratio of shared properties between a pair of match candidates to the total number of properties. RaQuN Scorer generates a file containing an array that captures all candidate pairs, each assigned with a corresponding similarity score. The inputs and the output of RaQuN Scorer are illustrated in Figure 3.17, while Figure 3.18 showcases the interface of the Python program used in this context.

```
> python .\raqun_scorer.py -h
        usage: raqun_scorer.py [-h] -invector _INVECTOR -knn _KNN -sprop [-o _OUTDIR]

        The purpose of the RaQuN Scorer is to find model candidate pairs rated with a score
        derived from the scoring function.
        It requires as inputs:
                - a .json with the set of vectorized models (output of vectorizer.py utility)
                - the number of nearest neighbors to use for searching the match candidates
                - the scoring (or similarity) function to evaluate the similarity of candidates
        It outputs:
                - A .json file with the set of model candidate pairs and their relevant similarity score

        options:
          -h, --help      show this help message and exit
          -invector _INVECTOR, --input-models-vector _INVECTOR
                          The absolute path to the .json file with the input models vector
          -knn _KNN, --k-nearest-neighbors _KNN
                          The number `k` of nearest neighbors to find for each point in the models vector
          -sprop, --scoring-shared-properties
                          Use shared properties as scoring function on the model matching
                          candidate pairs list
          -o _OUTDIR, --outdir _OUTDIR
                          The directory where the generated file will be stored.
                          If not defined, the current will be used.
```

FIGURE 3.18: RaQuN Scorer - Python help page

Following a similar approach to RaQuN Vectorizer, the implementation of RaQuN Scorer takes the form of a Python *class*, namely `ModelMatchScorer`, that provides APIs for executing the necessary operations of this software module. The class diagram in Figure 3.19 shows both the private members and the public interfaces of the class. Upon instantiation, the class requires the input JSON file, `models_vector.json`, containing the vectorized model elements. This data is used to construct two internal parallel lists: `point_vctrs`, which holds the point vectors, and `point_objs`, which stores the corresponding meta-data for each model element. By having these lists in parallel, we can easily access the data of each model element using the same index. The point vectors have to be stored separately in a dedicated list in order to leverage the `spatial.KDTree` library of the Python package `scipy` for creating the k-d tree.

The activity diagram presented in Figure 3.20 provides a visual representation of the execution flow of the RaQuN Scorer program. The process begins by verifying that the user has provided the necessary arguments, followed by the creation of an instance of the `ModelMatchScorer class`. Subsequently, the `get_neighbors()` API is employed which creates the k-d tree from the point vectors list by utilizing the `spatial.KDTree` library and using its function `query()` to find the closest neighbors with the Euclidean distance. The discovered neighbors are then stored within the meta-data internal list.

Note:

It should be noted that there may be multiple model element points sharing the same vector representation. In such cases, the RaQuN algorithm dictates the automatic adjustment of the number of neighbors to be searched. This adjustment applies in both cases where multiple neighbor points share the same vector representation, or when one or more neighbor point have the same vector represantation with the actual point. The implementation of the `get_neighbors()` function accounts for this logic.

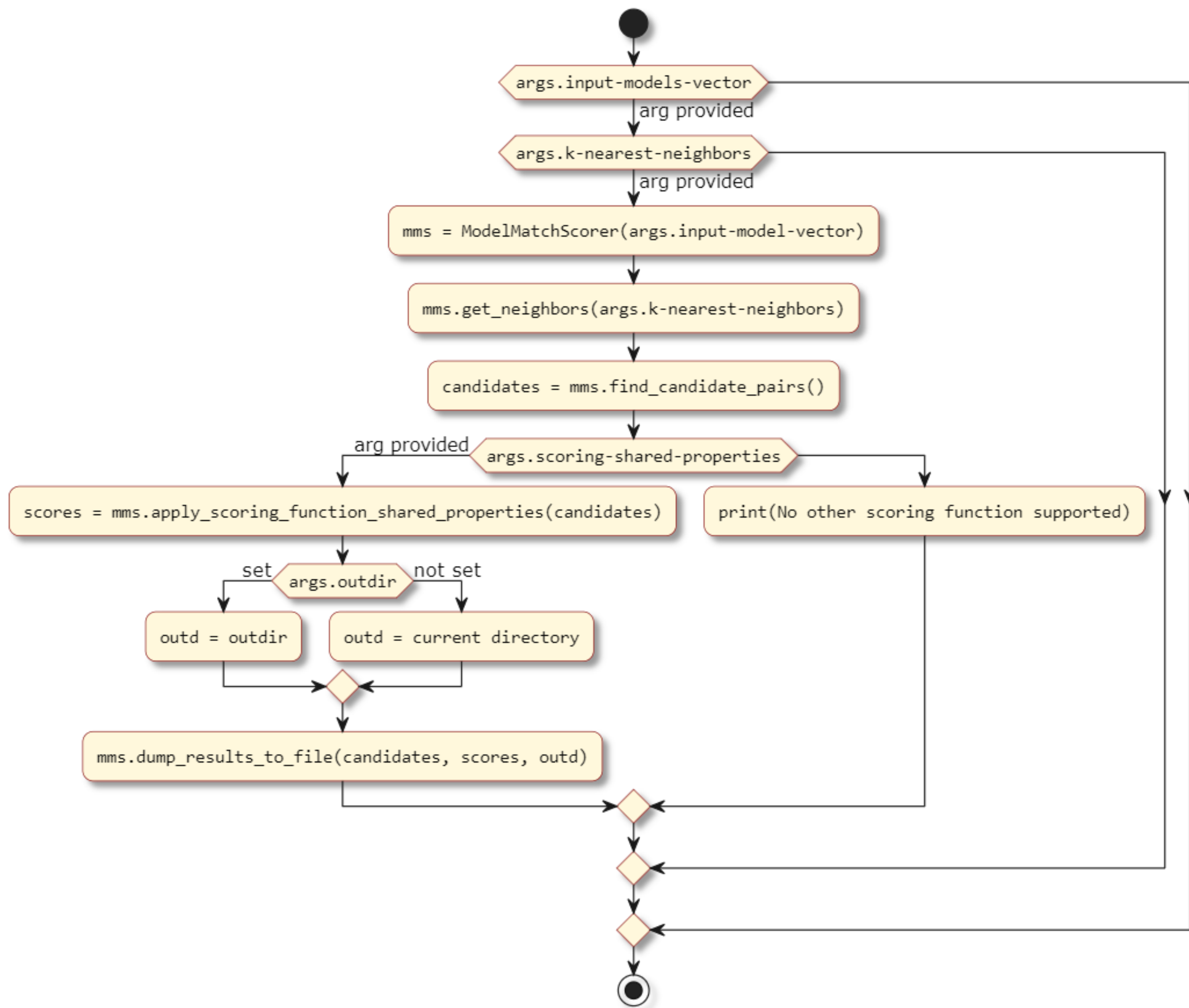

FIGURE 3.20: RaQuN Scorer - Activity Diagram

The subsequent step in the execution flow involves the `find_candidate_pairs()` function, which iterates through all model elements, creating pairs with each of their neighbors. According to the RaQuN algorithm, a pair of model elements is considered a valid candidate only if the elements belong to different models. Hence, the function filters out all pairs belonging to the same models, leaving only the valid candidate pairs. These pairs are returned as a list.

Then, RaQuN Scorer proceeds to assign scores to each candidate pair using the selected scoring function (`apply_scoring_function_shared_properties()`). This function calculates the similarity score for each candidate pair. The resulting list of scored candidate pairs is then outputted to a JSON-formatted file (Figure 3.22). Each entry of the `candidates` array in the generated file consists of the members `from_name`, `_from`, `to_name`, and `_to` indicating the identifier and the name of both model elements contained in the candidate pair. It is important to note that the terms "from" and "to" are utilized for seamless integration with the VARIOUS framework which uses them as keywords within the database infrastructure. However, their semantics should be disregarded, as they do not indicate a direction but rather a relationship of similarity.

```
> python .\raqun_scorer.py -invector .\models_vector.json -knn 3 -sprop -o .
        The candidates json file was generated successfully:
        .\models_candidates.json
```

FIGURE 3.21: RaQuN Scorer - Execution

Aligner

The implementation concludes with the RaQuN Aligner which is responsible for aligning the candidate pairs generated by the RaQuN Scorer to produce the final model match resutls. The RaQuN Aligner takes two essential inputs: the candidate pairs from the RaQuN Scorer and the comprehensive list of vectorized model elements generated by the RaQuN Vectorizer. This list serves as an initial baseline from which the alignment process gradually constructs the matches in the form of tuples. The user is able to influence the similarity tolerance by specifying a minimum threshold for the similarity score. This parameter can be dynamically set as an argument to the Python program. Figure 3.23 illustrates the inputs and outputs of RaQuN Aligner, while Figure 3.24 showcases the interface of the Python program with its help page.

The RaQuN Aligner is represented by the Python class `ModelMatchAligner`, as depicted in Figure 3.25. This class provides the necessary APIs to perform the alignment operation and generate the results file. The instantiation of the class requires the input of both `models_vector.json` and `models_candidates.json` whose data are stored in private arrays (`elements[]`, `candidate_pairs[]`).

The core functionality of the ModelMatchAligner is implemented in the public function `align_candidates_to_matches()`. This function carries out the alignment process and returns the final matches as a list of tuples. Each tuple may contain one or more model element IDs. If a tuple consists of only one element ID, it signifies that the respective model element did not find any similarity with other model elements. The aligmnent process begins by initializing the list of matches with single-element tuples extracted from the full list of vectorized elements. It then iterates through the given candidate pairs and merges their corresponding elements with the appropriate tuples. The user's input of minimum score is taken into consideration during this process. If a candidate pair has a score lower than the specified threshold, it is disregarded and not included in the alignment. The activity diagram of Figure 3.26 provides a visual representation of the execution flow.

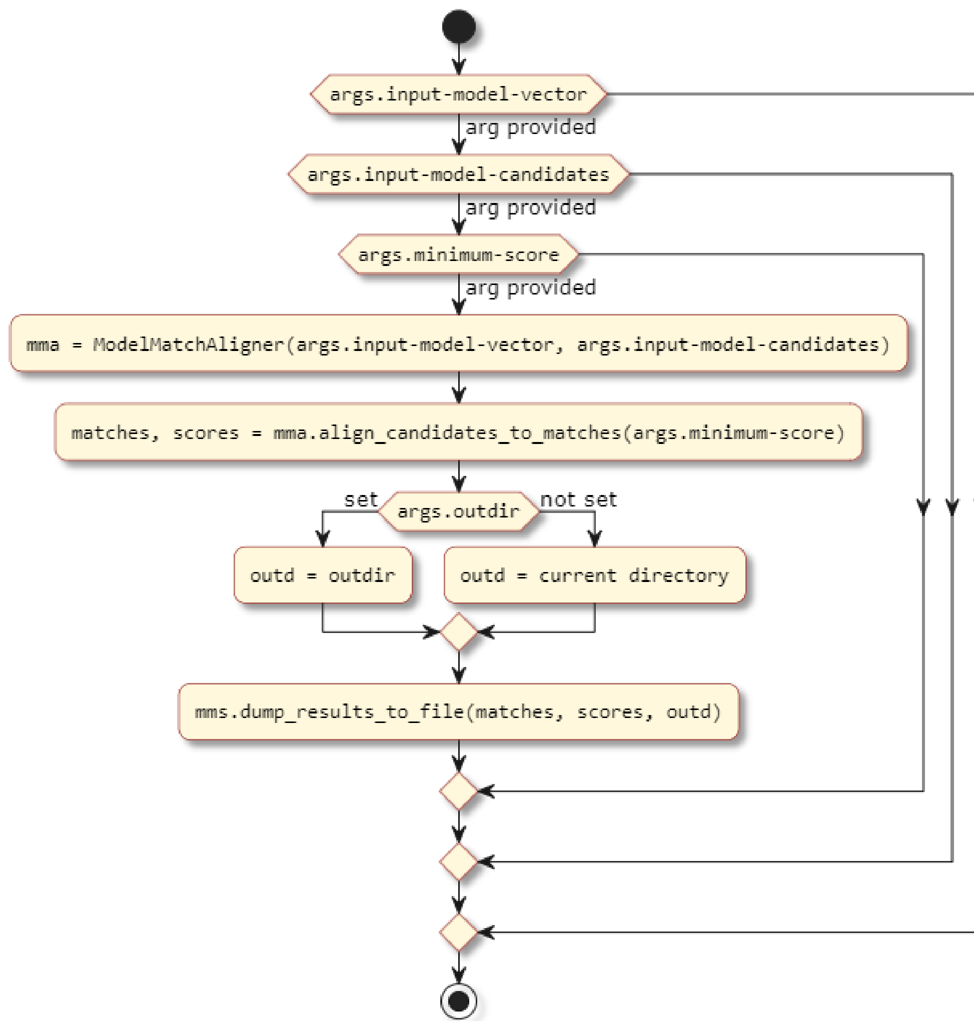

FIGURE 3.26: RaQuN Aligner - Activity Diagram

The successful execution of the RaQuN Aligner Python program is depicted in Figure 3.27, which demonstrates the outcome of the process. As a result, the program generates the JSON file `models_matches.json` containing the final list of match tuples (Figure 3.28). Each entry within the `matches` member represents a model match tuple, consisting of the IDs of the model elements that constitute the match along with the aggregated score of the involved candidate pairs. The element IDs are defined in the input file with models that is exported from the VARIOUS database (3.12). Finally, an overall score is added at the end of the file with the sum of all individual scores.

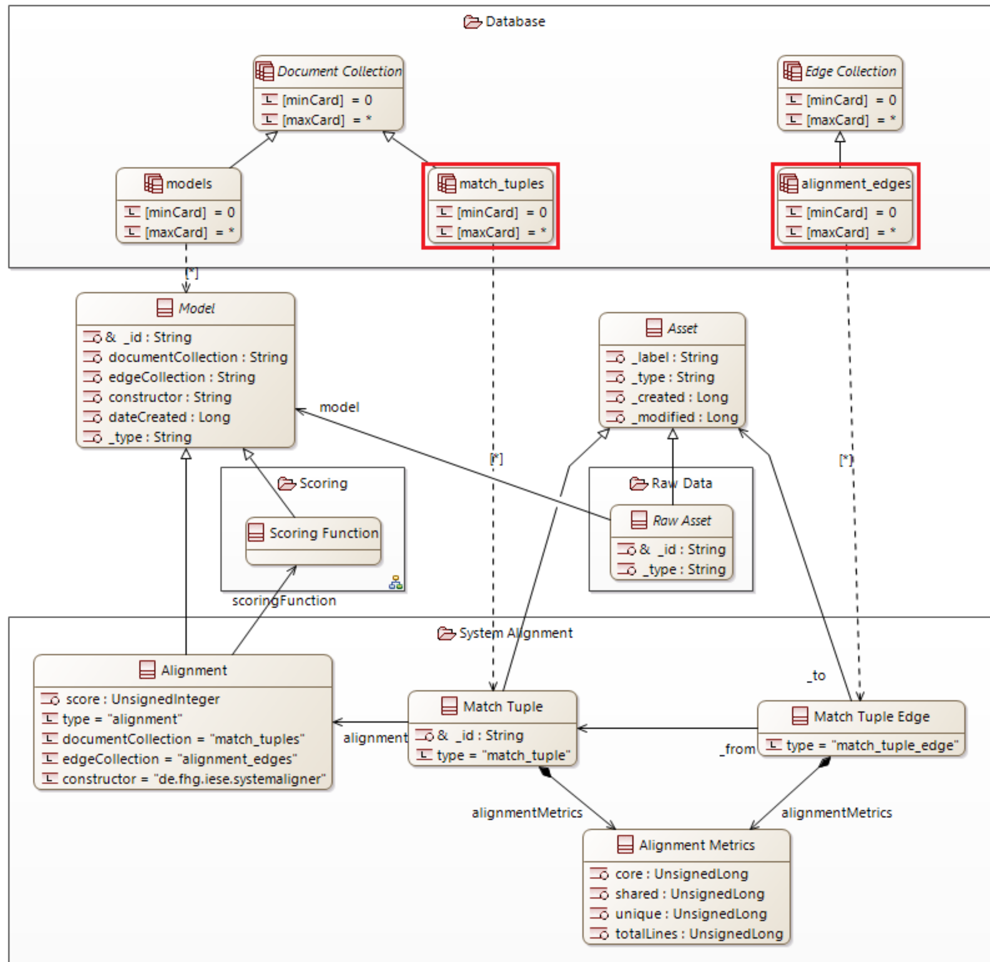

FIGURE 3.8: VARIOUS Database Class Diagram - Matches

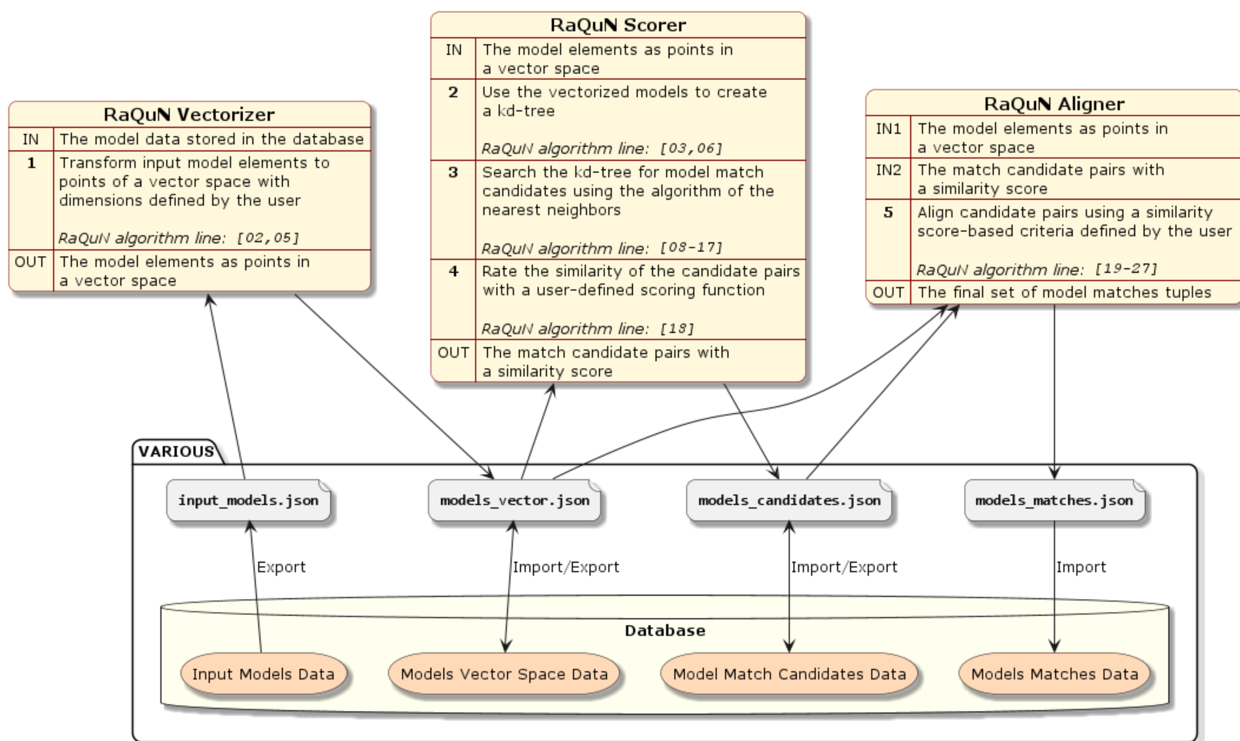

FIGURE 3.9: RaQuN implementation - High Level View

```
{
    "variants": [
        {
            "_id": "Model_A",
            "components": [
                {
                    "_id": "1",
                    "name": "History",
                    "type": "uml-element"
                },
                {
                    "_id": "2",
                    "name": "n_History",
                    "type": "uml-element-property"
                },
                {
                    "_id": "3",
                    "name": "procedure",
                    "type": "uml-element-property"
                },
                /* ... */
            ],
            "connections": [
                {
                    "_from":"1",
                    "_to":"2",
                    "type":"has"
                },
                {
                    "_from":"1",
                    "_to":"3",
                    "type":"has"
                },
                /* ... */
            ]
        },
        {
            "_id": "Model_B",
            "components": [/* ... */],
            "connections": [/* ... */]
        },
        /* ... more models ... */
    ]
}
```

FIGURE 3.12: RaQuN Vectorizer - Snapshot of `input_models.json`

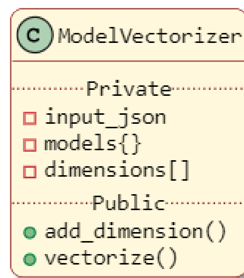

FIGURE 3.13: RaQuN Vectorizer - Class Diagram

```
{
    "input": ".\\input_models.json",
    "timestamp": "2023-06-19 16:57:53.490360",
    "dimensions": [
        {
            "dimension_id": "
                properties_average_name_length",
            "position": "0"
        },
        {
            "dimension_id": "number_of_properties",
            "position": "1"
        }
    ],
    "points": [
        {
            "element_id": "1",
            "model_id": "Model_A",
            "element_name": "History",
            "properties": [
                "n_History",
                "procedure",
                "patientProfile",
                "nurse",
                "physician"
            ],
            "vector": [
                9.2,
                5
            ]
        },
        /* ... more points ... */
    ]
}
```


FIGURE 3.16: RaQuN Vectorizer - Snapshot of `models_vector.json`

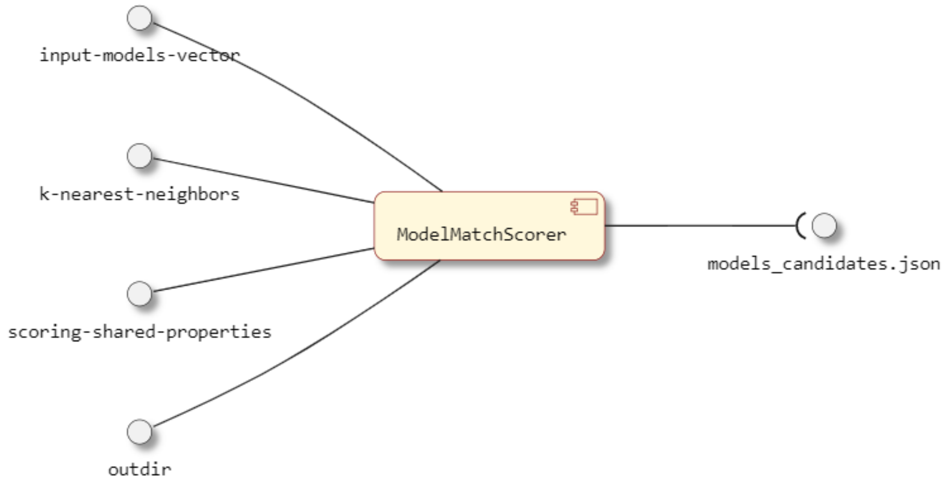

FIGURE 3.17: RaQuN Scorer - Component Diagram

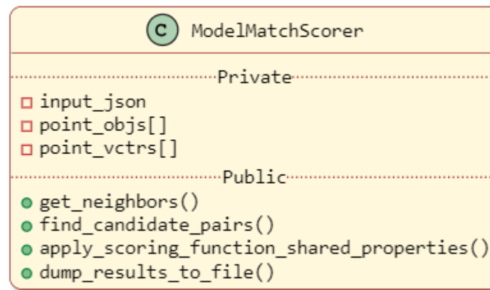

FIGURE 3.19: RaQuN Scorer - Class Diagram

```
{
    "input": "models_vector.json",
    "timestamp": "2023-03-26 17:08:13.328548",
    "candidates": [
        {
            "from_name": "Model_A_Physician",
            "_from": "7",
            "to_name": "Model_B_Physician",
            "_to": "17",
            "score": 0.6666666666666666
        },
        {
            "from_name": "Model_A_AdminAssistant",
            "_from": "13",
            "to_name": "Model_B_AdminAssistant",
            "_to": "30",
            "score": 0.5
        },
        {
            "from_name": "Model_A_AdminAssistant",
            "_from": "13",
            "to_name": "Model_C_AdminAssistant",
            "_to": "34",
            "score": 0.75
        },
        {
            "from_name": "Model_B_Ward",
            "_from": "23",
            "to_name": "Model_C_Unit",
            "_to": "39",
            "score": 0.09090909090909091
        },
        /* ... more candidate pairs ... */
    ]
}
```

FIGURE 3.22: RaQuN Scorer - Snapshot of `models_candidates.json`

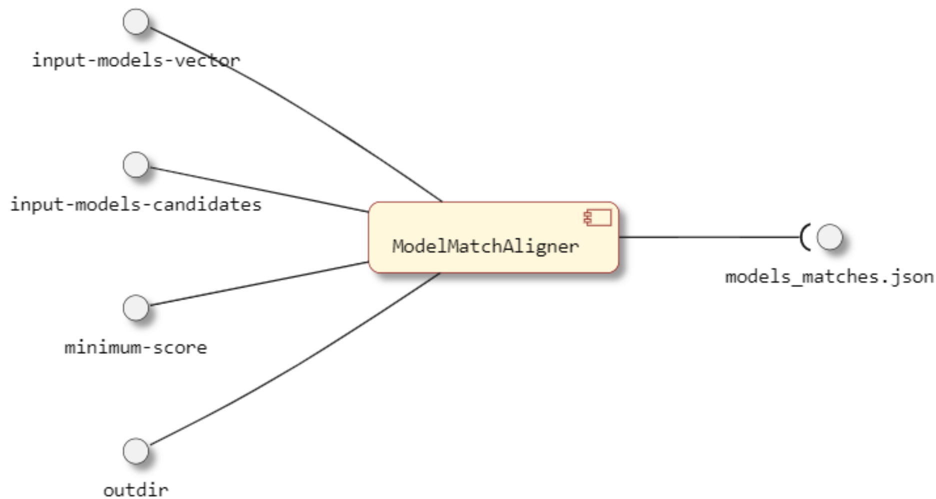

FIGURE 3.23: RaQuN Aligner - Component Diagram

```
> python .\raqun_aligner.py -h
        usage: raqun_aligner.py [-h] -invector _INVECTOR -incandidates _INCANDIDATES -minscore _MINSCORE
                                [-o _OUTDIR]

        The RaQuN Aligner is used to do the alignment of the model match candidate pairs
        generating a file with the final model match results. The matches are determined
        based on the user-defined minimum score limit.

        options:
        -h, --help            show this help message and exit
        -invector _INVECTOR, --input-models-vector _INVECTOR
                                The absolute path to the .json file with the input models vector
        -incandidates _INCANDIDATES, --input-models-candidates _INCANDIDATES
                                The absolute path to the .json file with the input models candidates
        -minscore _MINSCORE, --minimum-score _MINSCORE
                                The minimum score for a model candidate pair to be determined as a match
        -o _OUTDIR, --outdir _OUTDIR
                                The directory where the generated file will be stored.
                                If not defined, the current will be used.
```

FIGURE 3.24: RaQuN Aligner - Python help page

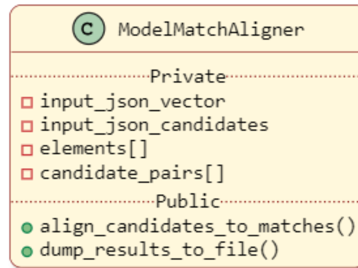

FIGURE 3.25: RaQuN Aligner - Class Diagram

```
> python .\raqun_aligner.py -invector .\models_vector.json -incandidates .\models_candidates.json
                            -minscore 0.5 -o .
        The matches json file was generated successfully:
        .\models_matches.json
```

FIGURE 3.27: RaQuN Aligner - Execution

```
{
       "input (elements)": ".\\models_vector.json",
       "input (candidates)": ".\\models_candidates.
            json",
       "timestamp": "2023-07-02 14:05:52.162061",
       "matches": [
              {
              "element_ids": [
                     "7",
                     "17"
              ],
              "score": 0.6666666666666666
              },
              {
              "element_ids": [
                     "13",
                     "30",
                     "34"
              ],
              "score": 1.25
              },
              {
              "element_ids": [
                     "1"
              ],
              "score": 0
              },
              {
              "element_ids": [
                     "23"
              ],
              "score": 0
              },
              {
              "element_ids": [
                     "39"
              ],
              "score": 0
              }
       ],
       "score": 1.9166666666666665
       }
```

FIGURE 3.28: RaQuN Aligner - Snapshot of `models_matches.json`

Chapter 4

Evaluation

This chapter focuses on addressing the *RQ3*. The RaQuN implementation discussed in the previous chapter will be evaluated having as reference the existing N-way model matching implementation of VARIOUS framework, the *System Aligner*. From now on, we refer to both implementations as algorithms.

4.1 Evaluation criteria and requirements

The evaluation criteria for assessing the performance of the algorimths are focused on the following aspects:

- *Efficiency*: This criterion evaluates the computational efficiency of the algorithms, which is measured by the execution time required to process the input data and generate results.
- *Accuracy*: This criterion assesses the similarity correctness of the matches generated by the algorithms.
- *Scalability*: This criterion measures the performance of the algorithms on large-scale input data.

To ensure a fair comparison between the algorithms, the following requirements must be met which are based on the aforementioned evaluation criteria:

- *Req.1*: Both algorithms should utilize the same dataset of input models.
- *Req.2*: The similarity function used in the two algorithms should be identical.
- *Req.3*: The measurement of the execution time and the accuracy of the matches should be precisely defined.

Regarding *Req.1*, we will utilize model data obtained from the VARIOUS framework database. Additionally, the experimental subjects [29] prepared by the authors of RaQuN

for evaluating the algorithm will be used. These datasets provide the model data for each subject in CSV format. Hence, it is necessary to convert them to a VARIOUS-friendly JSON format as illustrated in Figure 3.12. The Table 4.1 provides a summary of the data subjects included in our evaluation. The size of each dataset is determined based on the number of models and their contents. The column *Elements* indicate the average number of elements of each model, and the column *Properties* the average number of properties per element.

Subject	**Size**	**Models**	**Elements**	**Properties**
Hospitals	small	8	27.62	4.84
Warehouses	small	16	24.25	3.65
ApoGames	medium	20	63.05	19.62
ArgoUML	large	7	1752.86	9.05
ShopMngmt	very large	10	638.2	59.21

TABLE 4.1: Evaluation data subjects

To fulfill *Req.2*, the similarity function will be defined as the ratio of shared properties to all properties (Figure 4.1).

Similarity Score:

$$s_{pair} = \frac{|p_i \cap p_j|}{|p_i \cup p_j|}$$

→ s_{pair} : the similarity score of elements e_i and e_j
→ p_i : the properties of element e_i
→ p_j : the properties of element e_j

FIGURE 4.1: Similarity Score function in evaluation

Lastly, for the *Req.3*, the measurement of the execution time will be done starting from the user's trigger until the completion of result file generation. It is necessary to ensure that only the actual CPU time, that the respective process or thread consumes, is measured. Regarding the accuracy of the matches we will use the Prim's algorithm [30] modified for finding the maximum (instead of minimum) spanning tree of the undirected graph created by all possible pairs of a match tuple. The score per pair represents the weight of the edge

among the elements of the pair. The algorithm finds the subset of edges that form a tree which includes every vertex (i.e. element) of the match tuple, where the total weight of all the edges in the tree is maximized. Then we sum the weights (i.e. scores) of the found edges (i.e. pairs) to calculate the total score of the match tuple (Figure 4.2).

Accuracy Measure:

$$S_t = \sum_{i=1}^{m} s_{pair_i}$$

→ S_t : the total score of tuple t
→ m : the number of pairs found by Prim algorithm
→ s_{pair_i} : the score of pair i found by Prim algorithm

FIGURE 4.2: Accuracy Measure in evaluation

4.2 Meeting requirements with RaQuN implementation

The RaQuN implementation, as presented in Chapter 3, successfully satisfies the requirement for the similarity score function (*Req.2*) using the argument `scoring-shared-properties` (Figure 3.17, 3.18).

However, it needs some expansion to support the *Req.3* for achieving accurate measurements of execution time and matches quality. With regard to the time measurement, our implementation shall support the execution of all three modules on one run, and without generating the intermediate JSON files `models_vector.json` (Figure 3.16) and `models_candidates.json` (Figure 3.22) as the writing to a file takes a considerable amount of time increasing the CPU time significantly. Hence, we made the necessary changes to facilitate the transferring of data as objects in order to allow the one-off execution with no intermediate files. The measurement of the accuracy of the resulted matches requires the realization of Prim algorithm [30] to calculate the final score of each match tuple. The *RaQuN Aligner* module was further expanded to support this evaluation of matches.

Moreover, the vectorization of model elements that *RaQuN Vectorizer* implements is not suitable enough for the purpose of evaluation. This is because the dimensions it allows for selecting (Figure 3.10, 3.11) are not representative for measuring how similar (or near in the vector space) two elements are. Consequently, the implementation of Vectorizer needs to be adapted for performing the vectorization with a more meaningful selection of dimensions. RaQuN publication [9] discusses a domain-agnostic approach for selecting dimensions

of the vector space. This approach represents all distinct properties of model elements as dedicated dimensions in the vector space with range $0, 1^K$, where K denotes the number of disctinct properties across all elements. Each element can be represented within this vector space by assigning a value of 1 to the dimensions corresponding to the properties it possesses, while dimensions not applicable to the element are set to 0, indicating the absence of those properties. After incorporating this improved vectorization approach, our RaQuN implementation enhanced its ability to accurately represent the elements within the vector space, ensuring a more reliable evaluation process. The resulted generated JSON file of the reworked RaQuN Vectorizer is depicted in Figure 4.3.

Finally, it is important to note that, for the evaluation, the *RaQuN Scorer* will be configured to search for three (3) nearest neighbors in order to form the match candidates.

```
{
    "input": ".\\datasets\\input_models.json",
    "timestamp": "2023-07-04 11:11:21.972441",
    "dimensions": [
        {
            "dimension_id": "n_History",
            "position": "0"
        },
        {
            "dimension_id": "procedure",
            "position": "1"
        },
        {
            "dimension_id": "patientProfile",
            "position": "2"
        },
        /* ... more properties-dimensions ... */
    ],
    "points": [
        {
            "enum": "0",
            "element_id": "1",
            "model_id": "Model_A",
            "element_name": "History",
            "properties": [
                "n_History",
                "procedure",
                "patientProfile",
                "nurse",
                "physician"
            ],
            "vector": [
                1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
            ]
        },
        {
            "enum": "1",
            "element_id": "7",
            "model_id": "Model_A",
            "element_name": "Physician",
            "properties": [
                "n_Physician",
                "ex_medicalTeam",
                "patient",
                "ward",
                "history"
            ],
            "vector": [
                0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
            ]
        },
        /* ... more points ... */
    ]
}
```

FIGURE 4.3: Vectorization with properties as dimensions

4.3 Meeting requirements with System Aligner

The goal of RaQuN implementation, as part of the VARIOUS framework, was to keep System Aligner interfaces intact, in order to enable a smooth integration within the framework. Hence, there were no significant implementation changes needed for System Aligner as its implementation was already part of the VARIOUS ecosystem.

However, configuring the System Aligner to meet the evaluation requirements is necessary. *Req. 1* is already fulfilled, as the System Aligner is designed to work with datasets in the specified format (Figure 3.12). To satisfy *Req. 2*, the specified similarity function needs to be provided as input to the System Aligner. Lastly, the existing implementation of the System Aligner ensures compliance with *Req. 3*, which pertains to the accuracy measure.

4.4 Evaluation results

The results of our evaluation is presented in the following plots. For the execution time measurements, we used a box plot to visualize the variation in the data over 20 runs (Figure 4.4), and the quality of results was assessed using a bar plot, illustrating the achieved score levels for each algorithm across different experimental subjects (Figure 4.5). The subjects in both plots are arranged in order of their size. The experiments were conducted on a workstation equipped with an Intel(R) Core(TM) i7-10510U processor running at a frequency of 1.80 GHz.

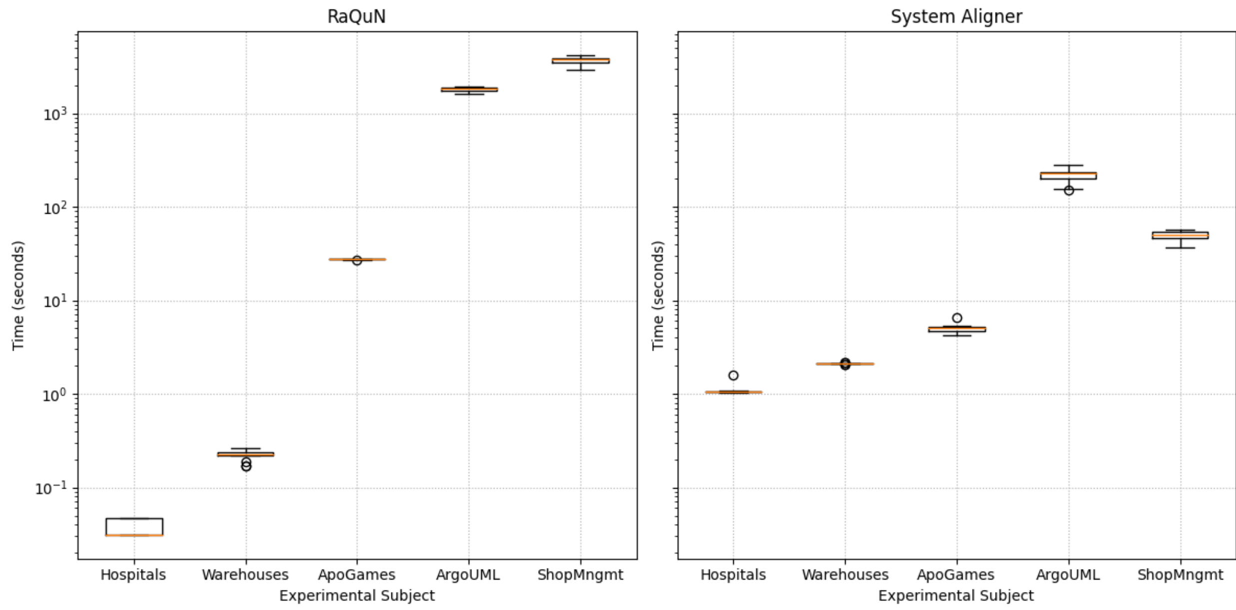

FIGURE 4.4: Execution Time Results Plot

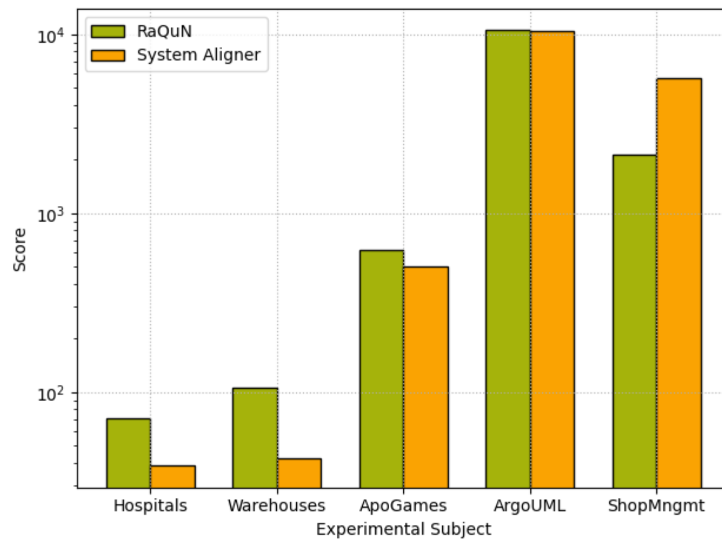

FIGURE 4.5: Matches Quality Results Plot

The evaluation results of the two algorithms revealed remarkable observations. In terms of execution time, RaQuN exhibited better performance in datasets with smaller sizes, such as *Hospitals* and *Warehouses*, while System Aligner showcased faster execution times in larger datasets (*ApoGames*, *ArgoUML*, and *ShopManagement*). Regarding the quality of results, RaQuN produced more accurate matches in datasets of smaller to medium sizes, while System Aligner outperformed RaQuN in larger-scale datasets.

The observed execution time results can be explained by considering the theoretical time complexities of the two algorithms. RaQuN's time complexity (Figure 2.3) has a greater growth rate for larger values of k than the Bioinformatics-based algorithm implemented in System Aligner (2.7), making it less efficient when the size of models increases. Moreover, the choice of implementing the two algorithms in different programming languages (Python for RaQuN and Java for System Aligner) may have contributed to the differences in the execution times, as the optimizations could have been uneven. Lastly, RaQuN implementation was designed aiming at high modularity which inevitably introduces some overhead on its performance.

Furthermore, the match scores results might have been affected by RaQuN's configuration for nearest neighbors searching. The evaluated implementation of RaQuN was set to find only three (3) nearest neighbors, which potentially limited the number of genuine match candidates that would lead to matches with higher scores. On the other hand, System Aligner is designed to construct larger match tuples, which could have contributed to its better performance in datasets with a higher number of elements and properties.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we conducted a qualitative evaluation of various N-way model matching approaches (*RQ1*). We compared the algorithms based on the accuracy, performance, scalability and configurability criteria. After careful analysis, we selected the "RaQuN" algorithm as the most promising candidate to implement within the product-line analysis framework "VARIOUS" (*RQ2*). The goal was to compare its efficiency with the existing mechanism of the framework known as "System Aligner".

The design and implementation of the RaQuN algorithm was grounded on ensuring seamless alignment with the interfaces of the System Aligner. Furthermore, we placed great emphasis on maintaining the overall behavior and compatibility between the two approaches. A key consideration during the development process was modularity, resulting in the division of the RaQuN implementation into three distinct modules to enhance reusability and maintainability.

Finally, we evaluated RaQuN and System Aligner having the efficiency, accuracy and scalability as the key criteria (*RQ3*). The comparison of the two approaches was done based on the same input model data and using identical similarity function. The evaluation provided valuable insights into the performance of RaQuN and System Aligner. RaQuN performed better in small-sized datasets, while System Aligner was superior in larger datasets. The observed differences in execution times and matches quality for diffferent dataset scales can be attributed to various factors, including algorithmic design, implementation choices, and dataset characteristics.

5.2 Future Work

The RaQuN algorithm shows high configurability exposing variation points for the user to tailor it to their own needs. In this thesis we mostly focused on the evaluation of the algorithm hence we implemented the variation points in a simplistic fashion aiming to have a fair comparison with the System Aligner of VARIOUS.

In the evaluation of the algorithm, the authors of RaQuN utilize the weight metric by Rubin and Chechik [8] for the similarity function variation point. This 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 with the formula of Figure 5.1.

$$w(t) = \frac{\sum_{2 \le j \le |t|} j^2 * n_j^p}{n^2 * |\pi(t)|}$$

- → $|t|$: the size of the match
- → n_j^p : the number of properties that occur in exactly j elements of the match
- → $\pi(t)$: the set of all distinct properties of all elements in t

FIGURE 5.1: Weight metric by Rubin and Chechik

Moreover, the RaQuN algorithm allows the user to further filter out unwanted matches by using the so-called *shouldMatch* function of the algorithm. This is applied on the matching phase of the algorithm where the match candidate pairs are evaluated before getting merged to the final match tuples. For our implementation we used the score limit that can be set by the user as an argument of the *RaQuN Aligner* module (Figure 3.24). However, the authors proposed a domain-agnostic metric where two matches t and t' are merged if the weight of the merged match $w(t \cup t')$ is greater than the sum of the individual match weights (Figure 5.2).

$$shouldMatch(t, t', e, e') := w(t \cup t') > w(t) + w(t')$$

FIGURE 5.2: Domain agnostic *shouldMatch* formula

It would be highly beneficial to conduct an investigation into the potential impact that these two aforementioned variation points could have on the overall accuracy of the

results exhibited by our current RaQuN implementation as well as to re-evaluate RaQuN's accuracy with larger number of nearest neighbors searching. Such an exploration would provide valuable knowledge that can further enhance the effectiveness of our RaQuN implementation in real-world scenarios.

Bibliography

[1] J. D. McGregor, L. M. Northrop, S. Jarrad, and K. Pohl, "Initiating software product lines," vol. 19, pp. 24–27, 2002, ISSN: 0740-7459. DOI: 10.1109/ms.2002.1020282.

[2] A. E. Chacón-Luna, A. M. Gutiérrez, J. A. Galindo, and D. Benavides, "Empirical software product line engineering: A systematic literature review," *Inf. Softw. Technol.*, vol. 128, p. 106 389, 2020. DOI: 10.1016/j.infsof.2020.106389.

[3] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh, "A manifesto for model merging," in *Proceedings of the 2006 international workshop on Global integrated model management*, ser. GaMMa '06, Shanghai, China: Association for Computing Machinery, May 2006, 5–12, ISBN: 1595934103. DOI: 10.1145/1138304.1138307. [Online]. Available: https://doi.org/10.1145/1138304.1138307.

[4] J. Rubin and M. Chechik, *Combining related products into product lines*, 2012. DOI: 10.1007/978-3-642-28872-2_20.

[5] F. A. Somogyi and M. Asztalos, "Systematic review of matching techniques used in model-driven methodologies," *Software and Systems Modeling*, vol. 19, no. 3, pp. 693–720, 2019. DOI: 10.1007/s10270-019-00760-x.

[6] Z. Xing and E. Stroulia, *Umldiff, An algorithm for object-oriented design differencing*, 2005. DOI: 10.1145/1101908.1101919.

[7] U. Kelter, J. Wehren, and J. Niere, "A generic difference algorithm for UML models," in *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*, P. Liggesmeyer, K. Pohl, and M. Goedicke, Eds., ser. LNI, vol. P-64, GI, 2005, pp. 105–116. [Online]. Available: https://dl.gi.de/20.500.12116/28304.

[8] J. Rubin and M. Chechik, "N-way model merging," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, Saint Petersburg, Russia: Association for Computing Machinery, Aug. 2013, 301–311, ISBN: 9781450322379. DOI: 10.1145/2491411.2491446. [Online]. Available: https://doi.org/10.1145/2491411.2491446.

[9] A. Schultheiß, P. M. Bittner, L. Grunske, T. Thüm, and T. Kehrer, "Scalable n-way model matching using multi-dimensional search trees," Fukuoka, Japan: IEEE, 2021, pp. 1–12, ISBN: 978-1-6654-3496-6. DOI: 10.1109/MODELS50736.2021.00010.

[10] V. L. Tenev, "Directed Coloured Multigraph Alignments for Variant Analysis of Software Systems," en, Bachelor Thesis, TU Kaiserslautern, Kaiserslautern, Nov. 2011.

[11] M.-S. Kasaei, M. Sharbaf, and B. Zamani, *A rule-based language for configurable n-way model matching*, 2022. DOI: 10.1109/iccke57176.2022.9960014.

[12] E. M. Arkin and R. Hassin, "On local search for weighted k-set packing," *Math. Oper. Res.*, vol. 23, no. 3, pp. 640–648, 1998. DOI: 10.1287/moor.23.3.640.

[13] B. Chandra and M. M. Halldórsson, "Greedy local improvement and weighted set packing approximation," *J. Algorithms*, vol. 39, no. 2, pp. 223–240, 2001. DOI: 10.1006/jagm.2000.1155.

[14] P. Berman, "A $d/2$ approximation for maximum weight independent set in d-claw free graphs," in *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, M. M. Halldórsson, Ed., ser. Lecture Notes in Computer Science, vol. 1851, Springer, 2000, pp. 214–219. DOI: 10.1007/3-540-44985-X_{1}{9}. [Online]. Available: https://doi.org/10.1007/3-540-44985-X_19.

[15] J. L. Bentley, "Multidimensional binary search trees used for associative searching," 509–517, Sep. 1975. DOI: 10.1145/361002.361007. [Online]. Available: https://doi.org/10.1145/361002.361007.

[16] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, *An exploratory study of cloning in industrial software product lines*, 2013. DOI: 10.1109/csmr.2013.13.

[17] A. Rouhi and B. Zamani, "Towards a formal model of patterns and pattern languages," vol. 79, pp. 1–16, 2016, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.06.002.

[18] V. L. Tenev and S. Duszynski, *Applying bioinformatics in the analysis of software variants*, 2012. DOI: 10.1109/icpc.2012.6240499.

[19] D. Gusfield, "Efficient methods for multiple sequence alignment with guaranteed error bounds," vol. 55, pp. 141–154, 1993, ISSN: 0092-8240. DOI: 10.1007/bf02460299.

[20] D. Reuling, M. Lochau, and U. Kelter, "From imprecise n-way model matching to precise n-way model merging.," *J. Object Technol.*, vol. 18, no. 2, 8:1, 2019, ISSN: 1660-1769. DOI: 10.5381/jot.2019.18.2.a8.

[21] D. Reuling, U. Kelter, J. Bürdek, and M. Lochau, "Automated n-way program merging for facilitating family-based analyses of variant-rich software," 1–59, Jul. 2019. DOI: 10.1145/3313789. [Online]. Available: https://doi.org/10.1145/3313789.

[22] J. Martinez, T. Ziadi, J. Klein, and Y. le Traon, *Identifying and visualising commonality and variability in model variants*, 2014. DOI: 10.1007/978-3-319-09195-2_8.

[23] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. L. Traon, *Automating the extraction of model-based software product lines from model variants (t)*, Lincoln, Nebraska, Nov. 2015. DOI: 10.1109/ase.2015.44. [Online]. Available: https://doi.org/10.1109/ASE.2015.44.

[24] T. M. Bulut, V. L. Tenev, and M. Becker, "Sys2vec: System-to-vector latent space mappings," 2021.

[25] T. M. Bulut, "Analysis of digital twin evolution via applyingartificial intelligence techniques," M.S. thesis, Technische Universität Kaiserslautern, 2020.

[26] V. L. Tenev and R. Martin, "Multi-Modell-Wissensgraph zur niederschwelligen datengestützten Entscheidungsunterstützung bei der Identifizierung von unwirtschaftlicher Variabilität," *Softwaretechnik-Trends*, vol. 43, no. 1, 2023.

[27] F. V. Latum, R. V. Solingen, M. Oivo, B. Hoisl, D. Rombach, and G. Ruhe, "Adopting gqm based measurement in an industrial environment," vol. 15, pp. 78–86, 1998, ISSN: 0740-7459. DOI: 10.1109/52.646887.

[28] J. Rowley, "The wisdom hierarchy: Representations of the dikw hierarchy," vol. 33, pp. 163–180, 2007, ISSN: 0165-5515. DOI: 10.1177/0165551506070706.

[29] A. Schultheiß, P. M. Bittner, and Timokehrer, *Alexanderschultheiss/raqun: Models 2021 - camera-ready*, 2021. DOI: 10.5281/ZENODO.5150388. [Online]. Available: https://zenodo.org/record/5150388.

[30] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, pp. 1389–1401, 6 1957, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1957.tb01515.x.