Karlsruhe Institute of Technology

Software Design and Quality

Entity Recognition in Software Documentation Using Trace Links to Informal Diagrams

Bachelor's Thesis of

Fabian Reinbold

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr.-Ing. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf H. Reussner
Advisor:	Dominik Fuchß, M.Sc.
Second advisor:	Sophie Corallo, M.Sc.

05. June 2023 – 05. October 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Maulbronn, 05.10.2023

. . . .

(Fabian Reinbold)

Abstract

Natural Language Software Architecture Documentation (NLSAD) and Software Architecture Model (SAM) provide information about a software systems design and qualities. Inconsistencies between these artifacts can negatively impact the comprehension and evolution of the system. ArDoCo is an approach that was proposed in prior work by Keim et al. to find such inconsistencies and relies on Traceability Link Recovery (TLR) between entities in the NLSAD and SAM. ArDoCo searches for Unmentioned Model Elements (UMEs) in the model and Missing Model Elements (MMEs) in the text using the linkage information. ArDoCo's approach shows promising results but has room for improvement regarding precision due to falsely identified textual entities. This work proposes using informal diagrams from the Software Architecture Documentation (SAD) to improve this. The approach performs an additional TLR between the textual entities and the diagram entities. According to heuristics, the linkage of textual entities and diagram entities is utilized to increase or decrease the confidence in textual entities. The Diagram Text TLR and its impact on ArDoCo's performance are evaluated separately using the same data set as previous work by Keim et al. The data set was extended to include informal diagrams. The Diagram Text TLR achieves a good F_1-score with Optical Character Recognition (OCR) of 0.54. The approach improves the MME detection (0.77→0.94 accuracy) by lowering the amount of falsely identified textual entities (0.39→0.69 precision) with a negligible impact on recall. The UME detection and ArDoCo's NLSAD to SAM are slightly positively impacted and continue to perform excellently. The results show that using informal diagrams to improve entity recognition in the text is promising. Room for improvement exists in dealing with issues related to OCR and diagram element processing.

Zusammenfassung

Natürlichsprachige Softwarearchitekturdokumentation (NLSAD) und Softwarearchitekturmodelle (SAM) enthalten Informationen über die Softwaresystemgestaltung und Qualitäten. Inkonsistenzen zwischen diesen Artefakten können das Verständnis oder die Evolution des Systems negativ beeinträchtigen. ArDoCo ist ein Ansatz, der in einer vorherigen Arbeit von Keim et al. vorgeschlagen wurde, um solche Inkonsistenzen zu finden und verwendet Nachverfolgbarkeitsverbindungen zwischen Entitäten in der NLSAD und dem SAM. Dabei sucht ArDoCo nach ungenannten Modellelementen (UMEs) im Modell und fehlenden Modellelementen (MMEs) im Text mithilfe der Verbindungsinformationen. ArDoCos Ansatz zeigt vielversprechende Resultate, aber auch Raum für Verbesserungen in Bezug auf die Präzision aufgrund von falsch identifizierten Textentitäten. Diese Arbeit schlägt die Verwendung von informellen Diagrammen aus der Softwarearchitekturdokumentation (SAD) vor, um das zu verbessern. Der Ansatz stellt zusätzliche Nachverfolgbarkeitsverbindungen zwischen den Textentitäten und den Diagrammentitäten her. Die Verbindung zwischen Textentitäten und Diagrammentitäten wird genutzt, um die Konfidenz in eine Textentität anhand von Heuristiken zu erhöhen oder zu verringern. Die Diagramm-zu-Text-Nachvollziehbarkeitsverbindungen und ihr Einfluss auf ArDoCos Leistungsfähigkeit werden separat mithilfe desselben Datensatzes wie in den vorherigen Arbeiten von Keim et al. evaluiert. Der Datensatz wurde um informelle Diagramme erweitert. Die Diagramm-zu-Text-Nachvollziehbarkeitsverbindungen erreichen ein gutes F_1-Maß unter Verwendung optischer Zeichenerkennung (OCR) in Höhe von 0.54. Der Ansatz verbessert die MME-Detektion (0.77⟶0.94 Genauigkeit) durch Verringern der Menge an falsch identifizierten Textentititäten (0.39⟶0.69 Präzision) mit einem vernachlässigbaren Einfluss auf die Sensitivität. Die UME-Detektion und ArDoCos NLSAD-SAM-Nachvollziehbarkeitsverbindungen werden leicht positiv beeinflusst und zeigen weiterhin eine exzellente Leistungsfähigkeit. Wie man an den Resultaten erkennen kann, ist die Verwendung von informellen Diagrammen zur Verbesserung der Entitätserkennung im Text vielversprechend. Raum für Verbesserungen herrscht im Umgang mit Problemen, die mit der OCR und dem Verarbeiten der Diagrammelemente zusammenhängen.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1

2 Foundations 3
- 2.1 Entity Recognition . . . 3
- 2.2 Similarity Metrics . . . 4
- 2.3 Traceability Link Recovery . . . 6
- 2.4 Architecture Documentation Consistency . . . 8
- 2.5 Benchmark Data Set . . . 10

3 Approach 13
- 3.1 Diagram Text Traceability Link Recovery . . . 13
- 3.2 Abbreviation Disambiguation . . . 14
- 3.3 Automated Diagram Extraction . . . 16
  - 3.3.1 Diagram Coverage . . . 16
  - 3.3.2 Homoglyphs and Confusables . . . 17
  - 3.3.3 Diagram Element Processing . . . 17

4 Implementation 21
- 4.1 Utility Classes . . . 21
- 4.2 Diagram Gold Standard . . . 26
- 4.3 Diagram Recognition Module . . . 30
  - 4.3.1 Diagram Recognition . . . 30
  - 4.3.2 Diagram Recognition Mock . . . 32
  - 4.3.3 Diagram-Backed Text State Strategy . . . 32
- 4.4 Diagram Connection Generator . . . 33
- 4.5 Diagram Inconsistency Checker . . . 34
- 4.6 Homoglyphs and Confusables . . . 36

5 Evaluation 39
- 5.1 Metrics . . . 39
- 5.2 Diagram-to-Sentence Trace Links . . . 42
- 5.3 Gold Standard Creation . . . 44
- 5.4 Diagram Text Traceability Link Recovery . . . 47
- 5.5 Impact on ArDoCo . . . 49
  - 5.5.1 SAD to SAM Traceability Link Recovery . . . 50

5.5.2 Missing Model Element Inconsistency Detection 53
5.5.3 Unmentioned Model Element Inconsistency Detection 57
5.6 Threats to Validity 59

6 Related Work 61

7 Conclusion And Future Work 63

Bibliography 65

List of Figures

2.1 Part-Of-Speech tagged sentence . . . 3
2.2 Example matchings of strings . . . 4
2.3 A *Levensthein distance* matrix for the two terms *ABCD* and *ABECG* . . . 5
2.4 Architecture Diagram of TeaStore (TS) [19] extended with an additional *Payment* node . . . 6
2.5 Architecture Documentation Consistency (ArDoCo) Traceability Link Recovery (TLR) approach [17] . . . 8
2.6 ArDoCo Inconsistency Detection approach [17] . . . 9

3.1 Extended ArDoCo Pipeline [17] . . . 13
3.2 Abbreviation Dictionary . . . 15
3.3 Image to Diagram Element Visualization . . . 18
3.4 Processing of a Diagram Element . . . 19

4.1 AbbreviationDisambiguationHelper cache structure . . . 23
4.2 Trace Link classes excerpt in UML . . . 27
4.3 DiagramProject structure excerpt in UML . . . 28
4.4 A block of confusables from the confusablesSummary file . . . 37

5.1 $2x2$ Confusion Matrix . . . 40
5.2 Formulas for TP, FP, TN, and FN . . . 40
5.3 Diagram-to-sentence Trace Links . . . 43
5.4 JSON schema in UML notation . . . 45

List of Tables

2.1 Text Entities 7
2.2 Diagram Elements 7
2.3 Established trace links 8

3.1 Impact of homoglyphs on similarity 17

5.1 Short description of the different trace types 46
5.2 Results for Diagram Text TLR with manually extracted Diagram Elements 48
5.3 Results for Diagram Text TLR with automatic Diagram Recognition . . . 48
5.4 Results for Architecture Document to Architecture Model TLR 52
5.5 Results for ArDoCo's Missing Model Element (MME) Inconsistency Detection 56
5.6 Results for ArDoCo's Unmentioned Model Element Inconsistency Detection 58

1 Introduction

Software architecture embeds the earliest design decisions of software systems and is vital in realizing and evolving the qualities of such systems [30]. A well-designed system architecture is central to creating and evolving high-quality software products and should therefore be the primary focus of software engineering [28].

Early manifestations of software architecture exist in the form of Software Architecture Documentations (SADs). Due to the quick evolution of SADs during development, inconsistencies between the different types of design documents can occur. Undetected inconsistencies can be problematic and have been the subject of prior work where an approach for Architecture Documentation Consistency (ArDoCo) was introduced [17]. So far, the implementation of this approach is limited to the detection of inconsistencies between the formal Software Architecture Model (SAM) and Natural Language Software Architecture Documentation (NLSAD). However, SADs can also contain informal diagrams, which do not adhere to more formal metamodels like the Unified Modelling Language (UML). Such informal diagrams consist of basic geometric shapes like boxes, lines, and text labels and contain information about the software architecture, including components and relationships between them. To find inconsistencies, ArDoCo uses a Traceability Link Recovery (TLR) approach to link model elements to the textual documentation. This requires a form of Natural Language Processing (NLP) to process the NLSAD and perform Entity Recognition (ER).

The motivation behind the proposed thesis is to investigate whether or not the informal diagrams can be used to improve the existing ER and to evaluate how such a change is reflected in the common TLR metrics that were used to evaluate the performance of ArDoCo in prior work [17]. According to the existing evaluation, there is still room for improvement regarding Missing Model Element (MME) inconsistencies [17]. A MME is a textual entity that was recognized in the NLSAD but does not appear in the SAM as a model element. The approach proposed by this work aims to improve the detection of MMEs by eliminating falsely identified textual entities. Reducing the number of falsely identified textual entities comes at the risk of reducing the number of correctly identified textual entities. The approach aims to minimize this effect.

To programmatically process the information in the diagrams, preprocessing with Optical Character Recognition (OCR) and Object Detection (OD) is required. The quality and completeness of the extracted diagram elements are also concerns that need to be evaluated. The existing prototype for the implementation goal of the proposed thesis is called Linking Sketches and Software Architecture (LiSSA) [10]. The goal of this thesis includes extending the current prototype to use the diagram elements that have been extracted by LiSSA, possibly improving the extraction of the diagram elements in the first place and linking them to the textual documentation that is extracted by ArDoCo.

2 Foundations

The following chapter introduces the knowledge this work is based on. The thesis closely relates to Traceability Link Recovery (TLR) and Entity Recognition (ER) automation problems. The Section 2.1 introduces the process of ER and gives an overview of established approaches. The TLR in this work is based on Similarity Metrics and the approach uses them for multiple processes. An overview of Similarity Metrics (SMs) is provided in Section 2.2. Subsequently, the concept of TLR is explained using an example in Section 2.3. This work builds on the inconsistency detection framework Architecture Documentation Consistency (ArDoCo), which uses ER and similarity-based TLR methods to establish trace links and entities to perform inconsistency detection. ArDoCo's structure is explained in Section 2.4. The data set that was previously used to evaluate ArDoCo is presented in Section 2.5.

2.1 Entity Recognition

The identification of model elements in Natural Language Software Architecture Documentation (NLSAD) is a special Named Entity Recognition (NER) problem [17]. NER is commonly used in Natural Language Processing (NLP) to extract *proper nouns* from unstructured text [25]. Proper nouns are nouns that refer to the name of a person, place, or thing [3]. NER can be broadly divided into three approaches: *Rules and templates*, *Machine learning*, and *Deep learning* [25].

NER based on rules and templates uses a set of manually constructed rules, templates, and heuristic algorithms [25]. Rules commonly use grammatical features with techniques such as Part Of Speech (POS) tagging, orthographic features including capitalization, and domain-specific dictionaries [26]. An example for a POS-tagged sentence is depicted in Figure 2.1. Templates consist of a pattern of POS tags. For example, the template "President *NN*" can be used in simple pattern matching with phrases like the one depicted to determine that "Eisenhower" is likely a proper noun. This approach can achieve high accuracy but often requires domain experts to create and customize rule sets [25]. The current ArDoCo implementation relies on such rule sets, which will be further explained in Section 2.4.

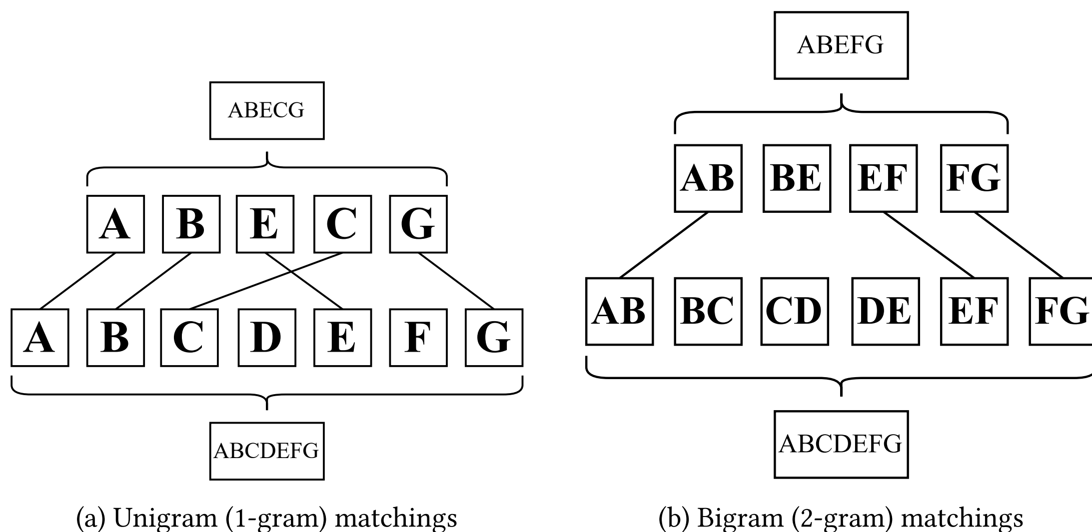
The	late	President	Eisenhower	was	an	avid	painter
DT	JJ	NN	NN	VBZ	DT	JJ	NN

Figure 2.1: Part-Of-Speech tagged sentence

Machine learning-based NER solves the issue of needing domain experts who manually create large rule sets. Instead, the approach uses training data to extract features that can be used to map real input data into feature space, where statistical methods can classify it into entities. However, this approach is usually restricted to supervised machine learning, which requires labeled training sets [26, 25]. This approach is made difficult by a lack of training data [17]. Known machine learning approaches include Hidden Markov Model, Conditional Random Field Model, Decision Trees, Maximum Entropy Models, as well as Support Vector Machines. They have been shown to achieve excellent results in reference to metrics such as F_1in some circumstances [33, 4]. Deep learning is a machine learning subfield that uses Artificial Neural Networks. Artificial Neural Networks can discover complex features automatically but require large sets of training data [24].

2.2 Similarity Metrics

SMs compare arbitrary strings and calculate their similarity or distance (dissimilarity). SMs are broadly divided into the three categories **String-Based**, **Corpus-Based** and **Knowledge-Based** [13]. This is relevant to this work because the approach proposed in ArDoCo creates trace links between *RecommendedInstances* and model elements that have equal or similar names [17]. Therefore, the choice of similarity measures heavily impacts the TLR metrics.



(a) Unigram (1-gram) matchings

(b) Bigram (2-gram) matchings

Figure 2.2: Example matchings of strings

String-Based SMs compare strings on character-level or term-level. ArDoCo currently uses *Jaro-Winkler similarity*, *Levenshtein distance* and *N-gram similarity*. *Jaro-Winkler similarity* is calculated using the sum of normalized matching characters in each string, the number of transpositions, and the length of a shared prefix [15]. A transposition occurs if a matching character is out of order. The score is linearly adjusted upwards depending on the length of a shared prefix [44]. An example is given in Figure 2.2a with five matches

and one transposition $EC \rightarrow CE$. The *Jaro similarity* is calculated as:

$$
\begin{aligned}
m &= 5 && \text{Matching characters} \\
s_1 &= ABECG && Term_A \\
s_2 &= ABCDEFG && Term_B \\
t &= 1 && \text{Transpositions} \\
j &= \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) && \text{Jaro similarity} \\
j &\approx 0.84 \\
prefix &= AB && \text{Shared longest prefix} \\
j_w &= j + 0.1|prefix|(1-j) && \text{Jaro-Winkler similarity} \\
j_w &\approx 0.87
\end{aligned}
$$

The *Levensthein distance* allows the character operations insertion, deletion, and substitution. It calculates the minimum number of operations needed to transform one string into another. Figure 2.3 shows the *Levensthein distances* for *ABCD* and *ABECG* with a calculated similarity of $sim_l = 0.6$ with $sim_l = 1 - \frac{d_{4,5}}{max(|s_1|,|s_2|)}$, $s_1 = ABCD$, $s_2 = ABECG$.

N-gram similarity works with substrings or terms of length *N*. *N-grams* define a family of word similarity measures, and some metrics, including the *Levensthein distance*, can be considered as special cases of this approach [23, 13]. As a mechanism, N-gram is capable of taking context into account. Like Winkler's approach, *N-gram* can take prefixes into account to emphasize the importance of earlier segments [23]. Figure 2.2b shows an example of bigram character matchings.

Corpus-Based SMs relies on the statistical distribution of words. It assumes that the semantic properties of words occurring in corpora, which are large text collections, can be inferred from the words surrounding it [32, 13]. Vector representations are commonly used with corpora to encode words explicitly or implicitly [32]. A modern approach for creating implicit encodings is word embeddings using Artificial Neural Networks (ANNs). A popular approach is called *Word2Vec* and consists of a two-layer neural network. Word2Vec can generate word and phrase embeddings and is available with pre-trained models in different languages [32]. An alternative approach is *GloVe*, which does not use ANN and performs similarly to *Word2Vec* using global co-occurrence information [32]. *Cosine similarity* is commonly used to determine the similarity between the vectors of two words and is defined as $sim_{cos}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| ||\vec{b}||}$.

		A	B	E	C	G
	0	1	2	3	4	5
A	1	0	1	2	3	4
B	2	1	0	1	2	3
C	3	2	1	1	1	2
D	4	3	2	2	2	2

Figure 2.3: A *Levensthein distance* matrix for the two terms *ABCD* and *ABECG*

Knowledge-Based SMs rely on lexical knowledge resources that can be seen as graphs where sets of synonyms are clustered, and edges represent semantic relations between

them [13]. For example, the words "Program" and "Application" can have the same sense and are therefore clustered. They are connected with a short path to "Software", which is semantically closely related. Popular lexical knowledge resources include *WordNet*, adapted versions of *Wikipedia* or *Wiktionary* and *BabelNet* [32]. SMs working on lexical knowledge resources are commonly based on the assumption that shorter paths between senses indicate similarity [32]. In the use case of ArDoCo *SEWordSim* provides a more accurate lexical knowledge resource than domain-unspecific resources like *WordNet* due to being constructed in a software context [42].

Before a SM is applied, words can be preprocessed with methods such as *stemming* and *lemmatization*. For example, the *WordNet* database is based on lemmas and removes inflections from queries [34]. The *SEWordSim* database only contains stemmed words [42]. The process of *stemming* tries to determine the word stem of any given word by removing common word suffixes such as "-ing" [31]. *Lemmatization* additionally takes inflections and POS into account. This can lead to more accurate results compared to *stemming* [38]. The word "meeting", as in a group of people socializing, might be stemmed to "meet" for example. By analyzing the context, *lemmatization* determines the proper base form. This base form is called lemma and keeps the sense of each word [38].

2.3 Traceability Link Recovery

Trace links can be used to discover the relationships and dependencies between different software artifacts [39]. If two elements from different artifacts represent the same entity, a trace link is established between them and indicates consistency [17]. Therefore, automatically creating trace links helps determine the consistency of Software Architecture Documentation (SAD). The following example illustrates TLR between text and a diagram. It is based on the open source software project TeaStore (TS) [19].

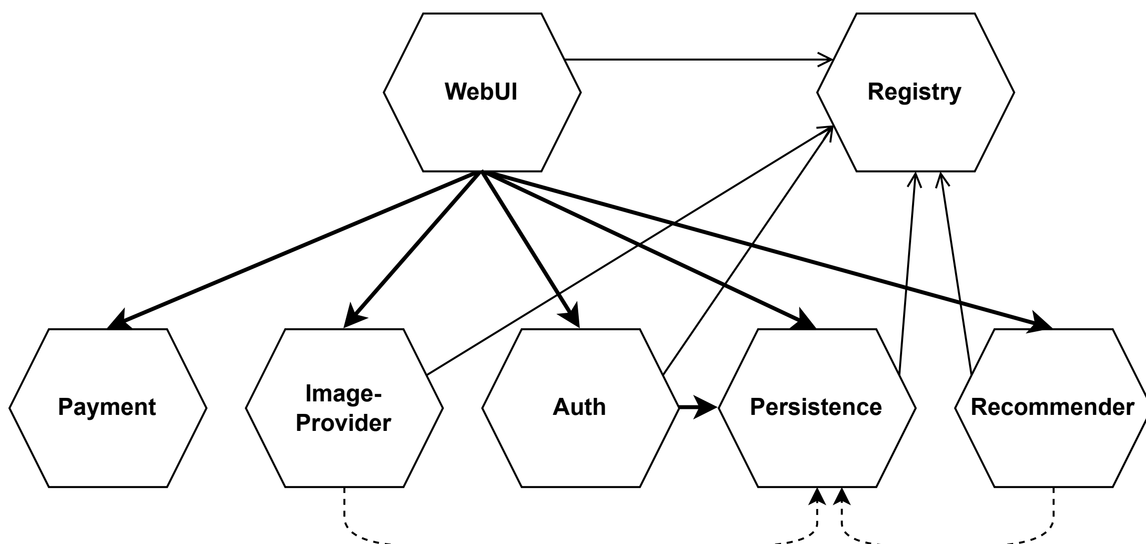


Figure 2.4: Architecture Diagram of TS [19] extended with an additional *Payment* node

The TLR is performed on the diagram in Figure 2.4 and the following two text fragments. The paper of Kistowski et al. [19] contains the first fragment. The second fragment describes a fabricated *Localization* service. Underlined words represent recognized entities.

- "All services communicate with the Registry. Additionally, the WebUI service issues calls to the Image-Provider, Authentication, Persistence and Recommender services."
- Additionally, the WebUI service calls the Localization service.

Before trace links between this NLSAD fragment and the informal diagram can be established, both artifacts must be processed to extract relevant elements. NER as described in Section 2.1 can be used to extract the underlined entities from the sentence. The diagram can be processed using a combination of Optical Character Recognition (OCR) and Object Detection (OD) techniques.

Name	**Type**
Authentication	service
Image-Provider	service
Localization	service
Persistence	service
Recommender	service
Registry	service
WebUI	service

Table 2.1: Text Entities

Name
Auth
Image-Provider
Payment
Persistence
Recommender
Registry
WebUI

Table 2.2: Diagram Elements

The tables 2.1 and 2.2 assume perfect NER and perfect diagram processing and are sorted in ascending lexicographical order for readability. In the next step, the set of text entities is compared to the set of diagram elements pairwise. This can be achieved by using various heuristics that rely on word similarity metrics and comparing the resulting similarity value towards predefined thresholds to determine an overall confidence. A trace link is created if the overall confidence exceeds a threshold. For this example, $p_{a,b}$ is defined as the longest prefix of the two words a and b. The similarity is calculated using $\frac{|p_{a,b}|}{max(|a|,|b|)}$. Under real conditions, ER and diagram processing are imperfect, which adds additional uncertainty to each element.

The text entity *Localization* constitutes a Missing Diagram Element (MDE) because it is not part of a trace link. Similarly, the diagram element *Payment* does not appear in the text, has no similarity to any other element, and is classified as an Unmentioned Diagram Element (UDE). No trace links are established for these elements, which indicates that the diagram and the text are inconsistent.

Text Entity	Diagram Element	Similarity
Authentication	Auth	0.29
Image-Provider	Image-Provider	1.00
Persistence	Persistence	1.00
Recommender	Recommender	1.00
Registry	Registry	1.00
WebUI	WebUI	1.00

Table 2.3: Established trace links

2.4 Architecture Documentation Consistency

ArDoCo is an approach for inconsistency detection between the NLSAD and the Software Architecture Model (SAM). ArDoCo builds on the NLSAD to SAM TLR approach SoftWare Architecture Text Trace link Recovery (SWATTR), which was first introduced by Keim et al. in 2021 [18, 17]. The approach detects inconsistencies using the linkage information gained during TLR. The framework uses a pipeline divided into multiple stages. The former SWATTR framework is integrated into ArDoCo's pipeline as depicted in Figure 2.5. NLSAD and a SAM are provided as input to ArDoCo and initially processed separately in the **Text Extraction** and **Model Provider** stages. Each pipeline stage consists of *Agents*, that run a variety of *Informants* to update the stage's *States*. Because ArDoCo is based on heuristics, elements are assigned confidence values, and configurable thresholds are used to discard elements.

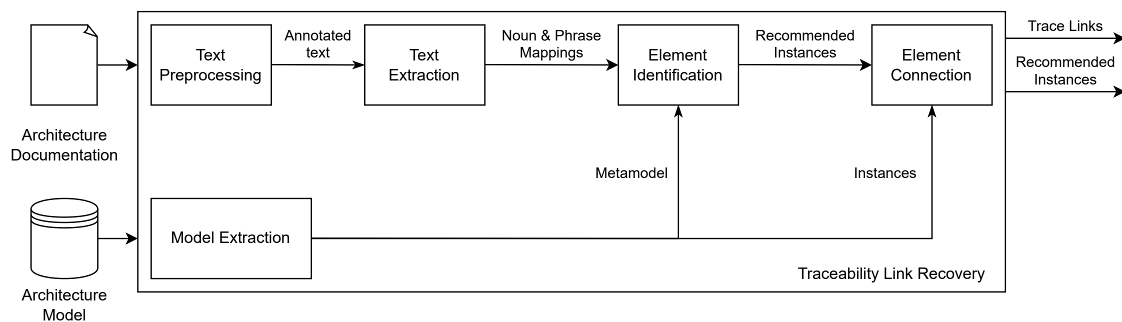


Figure 2.5: ArDoCo TLR approach [17]

The **Text Extraction** stage extracts elements from the text and clusters their mentions. The stage achieves this using various heuristics to extend the *TextExtractionState*. A rule- and template-based approach is used to perform NER and clustering on the text. The extracted information includes *NounMappings* and *PhraseMappings*. A *NounMapping* contains a reference and clusters mentions from the text, which are similar to the reference. A *PhraseMapping* consists of multiple consecutive words and represents a compound noun, such as “cache layer”. *PhraseMappings* are *n-grams* as described in Section 2.2.

During the **Model Provider** stage, ArDoCo uses a *Model Connector* for *Model Extraction*. This approach allows ArDoCo to handle different metamodels and to be expanded if no

appropriate connector exists. Currently, Unified Modelling Language (UML) and Palladio Component Model (PCM) in XML format, and a Java Code model in JSON are supported.

Element Identification is performed in the **Recommendation Generator**, as well as the **Connection Generator**. Each element is represented by a *RecommendedInstance*. A *RecommendedInstance* is equivalent to a candidate textual entity in this work. The creation of *RecommendedInstances* is based on heuristics. The **Recommendation Generator** stage uses the *NounMappings* and the metamodel. The **Recommendation Generator** stage initially searches for *Type NounMappings* that have an adjacent *Name-Or-Type* (*NORT*) or *Name* mapping. If such a combination is found, an additional check is needed to confirm that the *Type* is part of the metamodel. If that is the case, a recommended instance is created. Subsequently, this process is repeated with *PhraseMappings*. For example, the TS documentation mentions "Facade component" and "Component" is part of the PCM metamodel. Thus, a recommended instance is created with *Name* "Facade" and *Type* "Component" by ArDoCo.

The **Connection Generator** stage is responsible for creating the trace links between *RecommendedInstances* and model elements using SMs to compare their names. In its first step, it checks if any given text node is a *Name* or *Type* in the elements provided by the *Model Extraction*. Any such node is added as a *NounMapping* of the corresponding kind. Subsequently, *RecommendedInstances* are added for *NounMappings* similar to model elements. In an additional step, *RecommendedInstances* that mention the project name are removed because they are often mentioned in the NLSAD of the software and mistaken for named entities, which lowers the overall Precision [17]. This finalizes the set of *RecommendedInstances* proposed by ArDoCo. Trace links are now established by searching for model elements that are similar to *RecommendedInstances*. The probability of the *RecommendedInstance* is considered during trace link creation by comparison to a threshold.

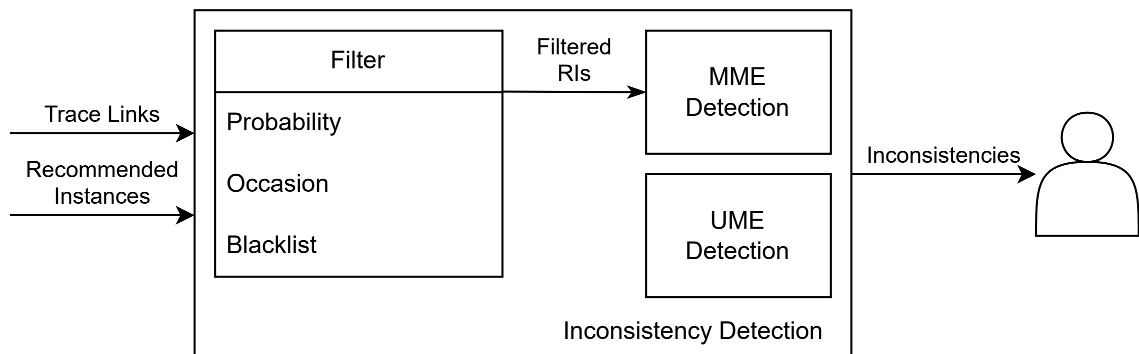


Figure 2.6: ArDoCo Inconsistency Detection approach [17]

ArDoCo's Inconsistency Detection approach is depicted in Figure 2.6. Initially, the **Inconsistency Detection** stage preprocesses the set of *RecommendedInstances* with different *Filter* heuristics. *RecommendedInstances* with a low probability are filtered out. A *RecommendedInstance* is also filtered if it only appears once in the text or contains a blacklisted word, such as computer- or software-related terminology. ArDoCo searches for Unmentioned Model Element (UME) and Missing Model Element (MME) inconsistencies. A Unmentioned Model Element (UME) inconsistency occurs if a model element has no cor-

responding textual entity. Since textual entities are represented by *RecommendedInstances* in this context, this is determined by whether a model element has at least one trace link to a filtered *RecommendedInstance*. A MME inconsistency occurs if a textual entity is not represented in the model. Missing Model Elements (MMEs) are found by searching the filtered *RecommendedInstances* for *RecommendedInstances* with no trace link.

2.5 Benchmark Data Set

This section introduces the data set used to evaluate ArDoCo. The information in this section originates from the paper in which the data set was published and the public repository [12, 11]. The Benchmark was initially published by Fuchß et al. as a data set for the research domain of Traceability Link Recovery (TLR) between Software Architecture Documentations (SADs) and Software Architecture Models (SAMs). The data set is based on the five projects MediaStore, TEAMMATES, BigBlueButton, TeaStore, and JabRef. The projects TEAMMATES, BigBlueButton, TeaStore, and JabRef each consist of a *Historical* and *Non-Historical* version. In this context, *Historical* refers to an older version of the projects being used. Because the data set was created for TLR between SAD and SAM, it contains a textual SAD, the SAMs, and a trace link gold standard for each project version. A PCM model and a UML component model are provided. PCM was chosen because it can provide different software architecture views. Additionally, UML component models were created to increase compatibility. The original data set has since been extended with an Unmentioned Model Element (UME) gold standard for each project version. A code model with corresponding gold standards has been added for some projects. The code model and code model-related gold standards are not further explained because this work is only concerned with SAD SAM TLR and inconsistency detection.

Each model element in the data set is assigned a Unique Identifier (UID). For example, the component "UI" with the UID "_1lMqsKESEeu-mYqkDskRow" is depicted in Listing 2.1. The amount of components contained in the models ranges from 6 to 14. The SAD is provided as a plain text file, where each line corresponds to a single sentence. Special characters, tables, figures, and captions have been removed from the text. The length of SAD ranges between 10 and 198 sentences.

The gold standards are saved as CSV files. For the SAD SAM TLR gold standards, the columns are the model element UID and a sentence number. For example, consider the first sentence from the TEAMMATES SAD text file: "Architecture contains UI Component,

```
<components__Repository
      xsi:type="repository:BasicComponent"
      id="_1lMqsKESEeu-mYqkDskRow"
      entityName="UI">
 ...
</components__Repository>
```

Listing 2.1: Example of a PCM component from the TEAMMATES repository

Logic Component, Storage Component, Common Component, Test Driver Component, E2E Component, Client Component." The sentence mentions the "UI" component from the previous PCM excerpt. Therefore, the gold standard for the SAD SAM TLR of TEAMMATES contains the row "_1lMqsKESEeu-mYqkDskRow,1". For UMEs, the gold standard file only needs to contain a list of model element UIDs.

3 Approach

The chapter introduces the approach of this work called *Entity Recognition in Software Documentation Using Trace Links to Informal Diagrams* (ERID). The thesis creates trace links between informal diagrams and candidate textual entities from the Natural Language Software Architecture Documentation (NLSAD). The process used to perform this linkage is described in Section 3.1. Abbreviations are problematic for a similarity-based Traceability Link Recovery (TLR) approach, especially for diagrams, where abbreviations may be more frequent due to space constraints. Section 3.2 introduces a method for disambiguating abbreviations. In a previous work called Linking Sketches and Software Architecture (LiSSA), an automatic diagram recognition was built and integrated into Architecture Documentation Consistency (ArDoCo). Section 3.3 describes how the approach adapts to automatic diagram recognition and mitigates associated problems.

3.1 Diagram Text Traceability Link Recovery

The initial step will focus on TLR between manually extracted diagram elements and textual entity candidates, represented by recommended instances. The approach currently implemented by ArDoCo is used to establish the trace links. The text will be processed by ArDoCo to extract recommended instances as described in Section 2.4. The diagram elements are temporarily transformed into model elements to create these trace links. This allows running ArDoCo's TLR, which is already capable of creating trace links between model elements and recommended instances. The Diagram Text TLR is encapsulated in the *Diagram Connection Generator* stage as depicted in Figure 3.1.

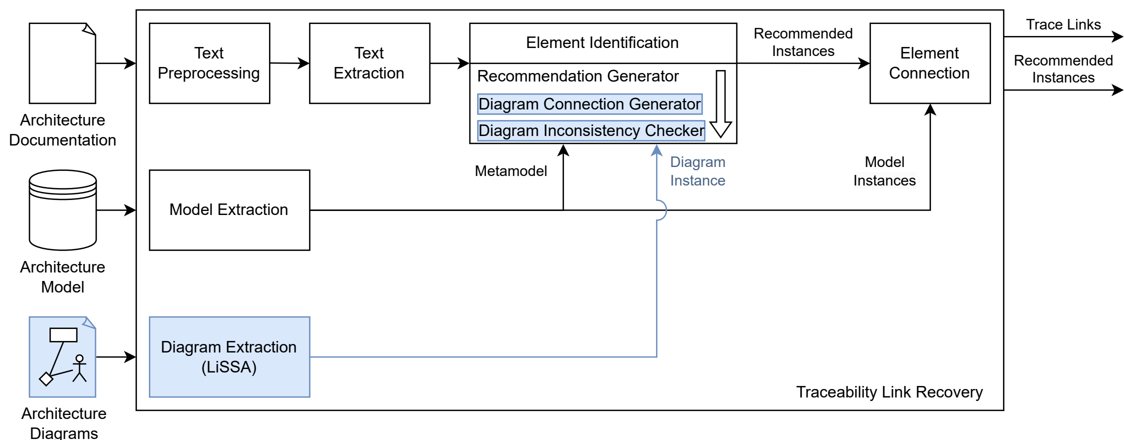


Figure 3.1: Extended ArDoCo Pipeline [17]

The trace links between diagram elements and recommended instances are passed along to the **Diagram Inconsistency Checker**. The **Diagram Inconsistency Checker** is responsible for finding Missing Diagram Element (MDE) and Unmentioned Diagram Element (UDE) inconsistencies using the linkage information. A MDE inconsistency occurs if a textual entity candidate is not contained in the diagrams. In contrast, the UDE inconsistency occurs if a diagram element has no corresponding textual entity candidate. The idea behind this approach is that the diagrams are closely related to the text. Therefore, actual textual entities are likely contained in the diagrams. If a textual entity candidate is not included in the diagram, the approach considers it less probable that the candidate is a textual entity.

Therefore, the **Diagram Inconsistency Checker** is responsible for affirming or disaffirming textual entity candidates, depending on whether they are consistent with the diagrams. To perform the affirmation of consistent textual entity candidates and the disaffirmation of inconsistent textual entities, the following heuristics are used:

H1 Negatively influence the probability of recommended instances contained in MDE inconsistencies

H2 Positively influence the probability of recommended instances that are consistent with the diagrams

The modified textual entity candidates are then passed along to the *Connection Generator* stage, where the pipeline continues as described in Section 2.4. ArDoCo uses confidence filters to remove recommended instances below a confidence threshold. Thus, the approach is a prefiltering step to the later stages of ArDoCo. To evaluate the Diagram Text TLR capabilities, the Benchmark data set presented in Section 2.5 is extended with diagrams and a gold standard for trace links between diagram elements and sentences. The creation of this gold standard is detailed in the evaluation chapter Section 5.3.

3.2 Abbreviation Disambiguation

In the context of this work, an abbreviation is a word that abbreviates an arbitrary amount of meanings. Abbreviations are problematic because they are a distinct alternative representation of their meaning. An abbreviation does not have to be used continuously. For example, “DB” may be used in a diagram and “DataBase” in the text. This is an example of discontinuous use of the “DB” abbreviation and can lead to no trace link between them. This can happen because the similarity between a word and its corresponding abbreviation may be low despite having the same meaning. For this reason, abbreviations should be considered in similarity-based TLR heuristics.

The abbreviation dictionary allows the use of the information from intra-artifact abbreviation disambiguation in inter-artifact processes such as TLR. Before this work, ArDoCo was incapable of any form of abbreviation disambiguation. The abbreviation disambiguation was originally conceived for the diagram elements processing step to aid the TLR between diagram elements and text entities. However, the Software Architecture Documentation (SAD) to Software Architecture Model (SAM) TLR suffers from the same issue.

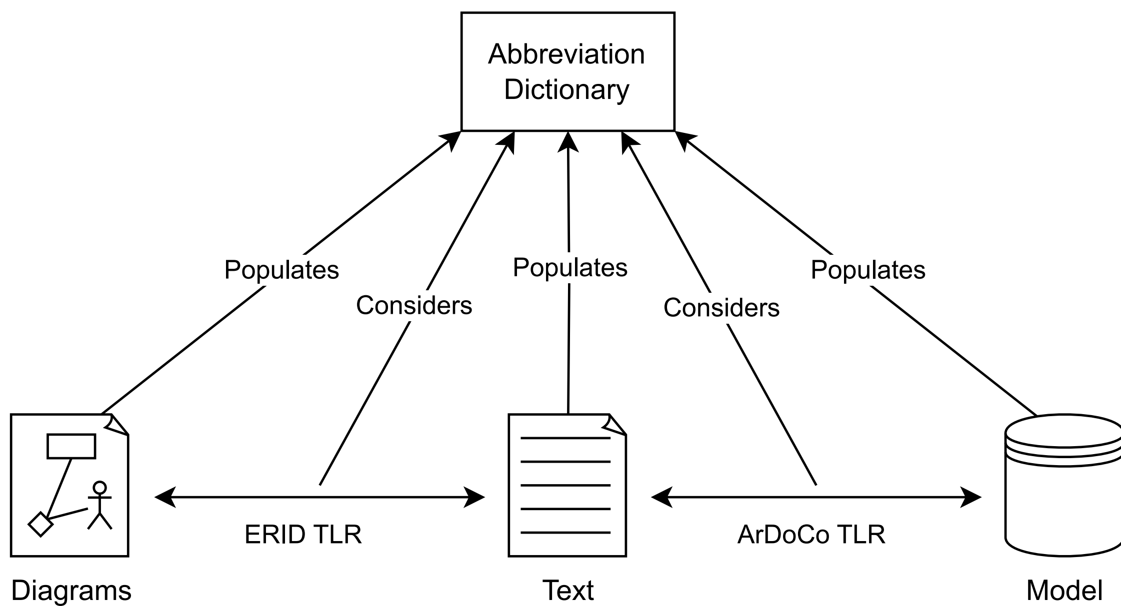


Figure 3.2: Abbreviation Dictionary

Therefore, the approach implements the abbreviation dictionary as a global solution. The abbreviation dictionary contains abbreviations and their associated meanings. Figure 3.2 shows how the dictionary is integrated into ArDoCo conceptually. It is a global dictionary that can be accessed and populated by all ArDoCo stages and from all available artifacts. Each entry consists of an abbreviation and an arbitrary amount of associated meanings. The abbreviation and the meanings are both strings, but a meaning may consist of multiple words. In some cases, the meaning of an abbreviation may be provided at its first use. For example, if the text contains the phrase "the DataBase (DB)", the abbreviation "DB" with meaning "DataBase" is added to the dictionary during the *Text Extraction* stage as displayed in Figure 3.2. If the model or a diagram also contains an element with the name "DB", the abbreviation dictionary can then disambiguate its name.

However, some abbreviations may be used without any further explanation. This is especially the case for abbreviations that are common terminology in the application field. An automated external dictionary lookup should be used to resolve such abbreviations, ideally with a dictionary close to the application field. This work proposes using external online abbreviation dictionaries as a fallback solution.

During the TLR processes, the abbreviation disambiguation can be used to improve the similarity-based trace link creation outcome. The similarity metrics need to be adjusted to consider abbreviations to achieve this. When comparing two terms $Term_A$ and $Term_B$, the abbreviations are taken into account by abbreviating all known meanings inside the terms. For example, assume that $Term_A = Database$, $Term_B = DB$ and "DB" is a known abbreviation of "Database". This pair of terms has a *Jaro-Winkler Similarity* $j_w \approx 0.59$. Even though they semantically refer to the same term, these terms will not be considered a match. However, if all meanings are substituted with the correct corresponding abbreviation, a

shared representation can be derived:

$$
\begin{aligned}
Term_A &= Database \\
Term_A &\xrightarrow[Database]{DB} Term'_A && \text{Substitution} \\
Term'_A &= DB \\
Term_B &= DB \\
Term_B &\xrightarrow[Database]{DB} Term'_B && \text{Substitution} \\
Term'_B &= DB \\
j_w(Term_A, Term_B) &\approx 0.59 \\
j_w(Term'_A, Term'_B) &= 1
\end{aligned}
$$

3.3 Automated Diagram Extraction

In the next step, the automatically generated diagram elements from the *Diagram Extraction* are used. Currently, these diagram elements are extracted with Optical Character Recognition (OCR) and Object Detection (OD) methods using the LiSSA prototype. Additional heuristics are required to deal with the extracted elements. Subsection 3.3.1 introduces a heuristic using the percentage of recommended instances covered by diagrams. The following section explains the impact of homoglyphs and confusables on the Similarity Metrics (SMs) and how to mitigate it. The last section covers how diagram elements are processed.

3.3.1 Diagram Coverage

If the diagrams cover only few textual entity candidates from the SAD, confidence adjustments need to be made to the heuristics used by the **Diagram Inconsistency Checker**. For example, if we rely on heuristic **H1** but have low diagram coverage, many *RecommendedInstances* may be disqualified, which can lead to more False Positives (FPs) in ArDoCo's Unmentioned Model Element (UME) detection. However, if the diagrams cover a large percentage of recommended instances, heuristic **H1** can be used with more confidence. To adjust to the diagram coverage, heuristic **D1** will be implemented.

D1 The impact of Missing Diagram Elements (MDEs) depends on the percentage of covered recommended instances

According to **H1**, the approach reduces the confidence in recommended instances, affected by a MDE inconsistency. In this case, the heuristic increases the reduction if the coverage is high. This is done because MDE inconsistencies are less common in the case of high diagram coverage. In contrast, the reward for consistency is highest if the coverage is low because it implies that the textual entity is particularly important.

3.3.2 Homoglyphs and Confusables

Characters like homoglyphs and Unicode *Confusables* pose an additional problem. Depending on the fonts, these characters can hardly be distinguished in diagrams because they are similar or identically looking [7]. This can cause a misread character by the OCR. An example of homoglyphs and Unicode *Confusables* characters are the letters A (U+0041), Α (U+0391), and А (U+0410), which look identical in the font of this document and many other fonts. This is intentional because they represent the same letter in the Latin, Greek, and Cyrillic alphabets. There are at least 7248 *Confusables* in Unicode[1]. The impact of such characters can be high depending on the SM as illustrated by Table 3.1.

Text Entity		**Diagram Element**		**Prefix**	**Jaro-Winkler**
Displayed	*Unicode*	*Displayed*	*Unicode*		
Authentication	U+0041	Auth	U+0041	Auth	0.86
Αuthentication	U+0391	Auth	U+0041	-	0.65

Table 3.1: Impact of homoglyphs on similarity

The *Jaro-Winkler Similarity* does not consider homoglyphs as matches. Because the characters do not match at the beginning, there is no shared prefix, and the similarity is merely based on the three matchings. This leads to a difference of 21 percentage points, which can falsely prevent the creation of a trace link. Heuristic **D2** is intended to reduce this effect.

D2 Positively influence the similarity of homoglyphs and confusables

3.3.3 Diagram Element Processing

Diagram elements are required to perform the TLR. In the context of this work, a diagram element is an element inside a diagram with a geometric shape. It may contain an arbitrary amount of text grouped into text boxes according to color and proximity. Each text box has a bounding box. However, a text box alone does not constitute a diagram element because it is merely text without a geometric shape. The text box therefore belongs to its parent diagram element or the diagram itself if it is not contained within any element. Diagram elements can be nested, in which case the hierarchical information may also be used during TLR. Boxes are primitive diagram elements with a box shape. These boxes can be automatically extracted from images using LiSSA and are the focus of this thesis. However, the heuristics used are not necessarily constrained to a box shape. Figure 3.3 shows two nested diagram elements with box shapes. The outer diagram element contains the two text boxes *DBUtility* and *CRUD* with different dominating colors. The inner diagram element includes a single text box *DBConnector*.

Due to the informal nature of diagrams, an approach is required to extract useful information from a diagram element before performing TLR. The TLR process is similarity-based and therefore requires comparisons between the texts within diagram elements and

[1] See unicode.org/Public/security/latest/confusablesSummary.txt

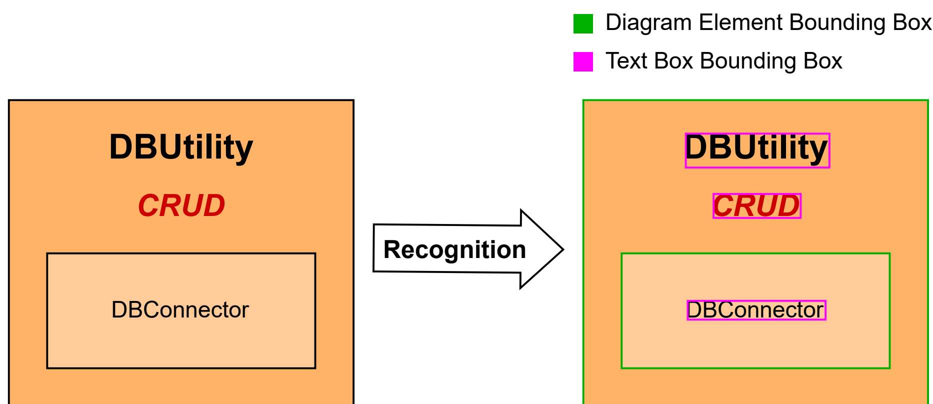


Figure 3.3: Image to Diagram Element Visualization

the references of textual entities. The diagram element text is searched for textual entity references to achieve this. However, the text boxes may contain information not specific to the diagram element, which can lead to FPs. An example of such information is the *CRUD* text box of the outer element in Figure 3.3. Therefore, the approach tries to retain as much useful text as possible while discarding as much useless text as possible. The textual entity references are singular words or short phrases. For this reason, the text boxes need to be broken up into similar-sized chunks. Overall, the text box preprocessing contains the following steps:

- Splitting the text
- Disambiguating abbreviations in the text
- Filtering out computer- and software-related words
- Finding references to the model

Initially, the text needs to be split into possible references. This includes splitting the text at enumeration separators and brackets. An enumeration is not a reference by itself but may contain multiple textual entity references as items. Consider the text box with the text "ConnectionManager, SQLiteConnector, QueryFactory" and the textual entity with reference "SQLiteConnector". If the approach were to compare the entire string to the textual entity reference directly, a character-based SM would punish the similarity score due to the unrelated preceding and subsequent terms. This would be opposed to the goal of finding mentions of textual entities in the text box and, consequently, the parent diagram element. Therefore, enumerations should be split up into their items. Brackets are processed separately based on the assumption that they might contain an alternative representation of the term before them.

Based on the assumption that a diagram in a SAD document only has limited space, abbreviations might be used to reduce the size of the text. However, abbreviations introduce ambiguity to a text and cause problems for the existing TLR approach as explained in Section 3.2. Therefore, abbreviations need to be identified and disambiguated. The proposed

heuristic to identify abbreviations in diagram elements is analyzing the letter case of (sub-)words. For example, if a *camelCase* word contains a sequence of two or more uppercase letters, they likely belong to an abbreviation. Occasionally, abbreviations deviate from this pattern by containing lowercase letters (e.g., ArDoCo, DoS - Denial of Service). This is mitigated by introducing a distance rule allowing up to n lowercase letters between uppercase letters. The maximum distance has to be carefully chosen to avoid falsely classifying ordinary words as abbreviations. This work will use $n = 1$. The proposed method cannot resolve lowercase abbreviations, which is a possible area for improvement in future work. The abbreviation disambiguation is stored in a global abbreviation dictionary. Section 3.2 explains how the stored abbreviation is disambiguated.

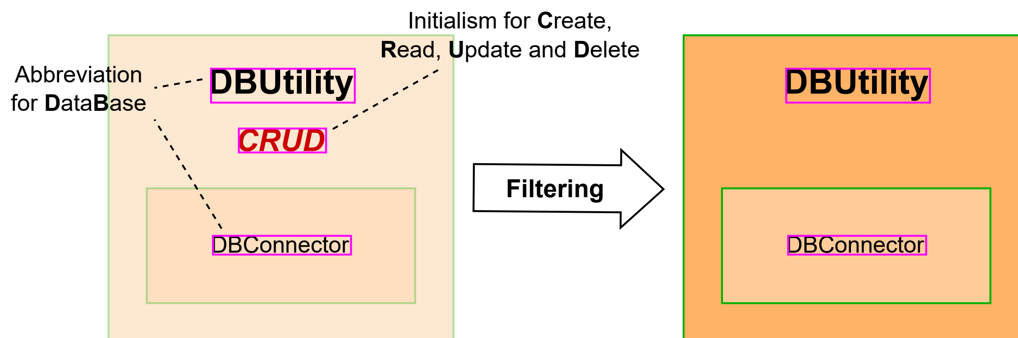


Figure 3.4: Processing of a Diagram Element

The filtering step tries to remove unspecific computer- and software-related terminology while preserving the specific parts of a text box. This step was introduced based on the observation that diagrams often contain additional texts describing the general technology used to implement some process. For example, the initialism *CRUD* in Figure 3.4 is a common initialism used to describe the four basic database operations and is not specific to the component *DBUtility*. Such words can be filtered using blacklists, which already exist within ArDoCo. However, these blacklists are manually created, and additionally considering the diagrams may cause considerable overhead for large projects. Therefore, automatically extracting blacklists from the online database *DBpedia* is proposed.

As a final step, references to the model are searched in the text boxes. This is based on the observation that a diagram element may be an informal representation of a model instance. Similar model instances are found by applying ArDoCo's similarity metrics to the text box references and the model instance references. The most similar model instance is chosen in conjunction with a similarity threshold. If no model instance is found, the diagram element remains unmodified. However, if a model instance is found, all references except the reference to the model instance are removed. This preserves the likely identity of the diagram element and removes any additional text, which could help reduce FPs during the diagram element to text TLR.

4 Implementation

This chapter presents the implementation of the conception. The shared functionality of multiple stages, agents, and informants is implemented in utility classes. These classes are listed and explained in Section 4.1. Section 4.2 explains the creation of the module handling the parsing of the gold standard and how issues with the existing gold standard were mitigated. The changes and extensions to the diagram recognition are detailed in Section 4.3 and its subsections. Subsequently, Section 4.4 describes the states, informants, and agents of the *Diagram Connection Generator* stage. The implementation of the *Diagram Inconsistency Checker* stage is presented in Section 4.5. In Section 4.6, the implementation of the homoglyphs and confusables is explained. *Italic* text refers to a class, record, enum, or interface in this chapter.

4.1 Utility Classes

The implemented utility classes are enumerated and explained in this section. Utility classes bundle functionality used by multiple stages, agents, informants, or globally. All utility classes are implemented in ArDoCo's *common* module.

FileBasedCache The two classes *FileBasedCache* and *SerializableFileBasedCache* are implemented to allow caching. The *FileBasedCache* is an abstract class that implements a cache backed by a single file. It provides functionality to retrieve content, cache content, and reset the file. The cache file is saved to an arbitrary subdirectory in the user data directory ArDoCo folder. The user data directory depends on the operating system. For example, "AppData\Roaming" is the user data directory in the case of Windows. The serialization and deserialization are delegated to implementing subclasses to allow for flexibility. The *FileBasedCache* is an *AutoCloseable* and automatically writes the cache to a file if changes occurred before closing it.

SerializableFileBasedCache *SerializableFileBasedCache* is an implementation of *FileBasedCache* based on Java's default serialization. This cache can be used with any object that supports the default serialization and is used to store information that is not required to be a humanly readable file. For example, the *SerializableFileBasedCache* can be used to cache the results of a previous test or evaluation run, which allows for a direct, in-depth comparison of the runs. ArDoCo can calculate metrics and provides text files containing information about a run. However, comparing a run by comparing the metrics gives little information about the cause and effect of a change. Comparing text files is laborious and provides limited insight because not everything can be logged.

Data such as known abbreviations in a text or a list of computer- and software-related words may be distributed as part of the Benchmark data sets. This is achieved by *FileBased-*

Cache implementations using JSON files and JSON serialization. The two implementations *AbbreviationDisambiguationHelper* and *DBPediaHelper* are based on a JSON *FileBasedCache* and are explained below.

AbbreviationDisambiguationHelper The *AbbreviationDisambiguationHelper* contains the abbreviations cache and abbreviation utility functions. The cache contains an arbitrary amount of *Disambiguation* instances. A *Disambiguation* instance consists of an abbreviation as a string and a set of associated meanings as strings. It is serialized into a JSON file containing a JSON array of JSON objects, where each JSON object corresponds to a *Disambiguation*. For example, Listing 4.1 shows the JSON file of an abbreviation cache with the two known abbreviations “GAE” and “UI”, as well as associated meanings.

The *AbbreviationDisambiguationHelper* allows disambiguating an abbreviation using the local cache and lookups in online abbreviation dictionaries. The cache is divided into a persistent (file-based) cache and a transient cache. Figure 4.1 shows the division of the cache. The persistent cache is populated using the online lookup. The transient cache is populated from definitions in the Software Architecture Documentation (SAD). For example, “..the Database (DB) is..” provides a definition of the abbreviation “DB” with meaning “Database”.

```
1  [
2      {
3        "abbreviation" : "GAE",
4        "meanings" : [
5          "Google App Engine", "Galaxy Advanced Engineering"
6        ]
7      },
8      {
9        "abbreviation" : "UI",
10       "meanings" : [
11         "User Interface", "Unemployment Insurance"
12       ]
13     }
14 ]
```

Listing 4.1: Example of an abbreviation cache JSON file

As part of this work, the Text Extraction stage is extended with the capability of searching for patterns such as *Meaning (Abbreviation)* and *Abbreviation (Meaning)* in the Natural Language Software Architecture Documentation (NLSAD) and adding these disambiguations to the transient cache. The transient cache is used because these disambiguations are specific to a project and should not be saved between runs. This functionality is implemented in the *AbbreviationAgent* and *AbbreviationInformant*. Potential abbreviations are identified using the *couldBeAbbreviation* method of the *AbbreviationDisambiguationHelper*, based on the percentage share of uppercase characters and a threshold. The neighboring words are searched for the sequence of characters in the abbreviation to determine a

meaning. Using the previous example, "Database" would be determined as a meaning for the abbreviation "DB" because it contains the characters of the abbreviation in order.

The meaning of the abbreviation is allowed to stretch across multiple words until a configurable distance limit is reached. This process can determine multiple candidate meanings for an abbreviation if the sequence of characters occurs multiple times in the neighboring words. The sequence distribution is irrelevant as long as it is in order. An exception is the first character of the abbreviation, which has to be the first character of any candidate meaning. To find the proper meaning, a heuristic calculates a score for each candidate meaning. This heuristic compares the candidate meaning and the abbreviation. It rewards the three features: *Any match*, *case match*, and *initial match*. For the previous example "..the Database (DB) is..", both "D" and "b" are rewarded for *any match*. "D" is additionally rewarded for *case match* and *initial match* because its casing matches the abbreviation, and the match is at the start of a word.

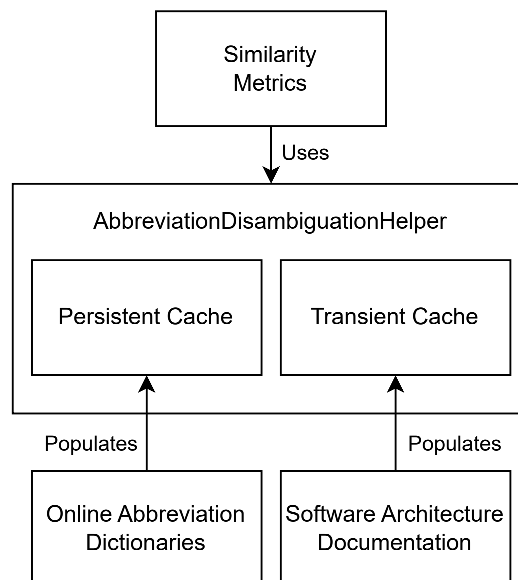


Figure 4.1: AbbreviationDisambiguationHelper cache structure

In addition to the previously mentioned *couldBeAbbreviation* method, the *AbbreviationDisambiguationHelper* provides a *getAbbreviationCandidates* method. This method takes a string parameter and applies a regular expression to it that matches sequences of uppercase characters with up to one lowercase character between them. The regular expression takes *CamelCase* into account. It uses a lookahead to support abbreviations at the word boundaries and abbreviations inside camel-cased phrases. For example, *getAbbreviationCandidates("DBLoremDDoSIpsumID")* returns $\{DB, DDoS, ID\}$, rather than $\{DBL, DDoSI, ID\}$.

If a potential abbreviation has to be disambiguated, the *AbbreviationDisambiguationHelper* first checks whether it is in the cache. If a matching *Disambiguation* exists, the associated meanings are returned. The transient cache has precedence to ensure that available project-specific definitions are used. Alternatively, the two abbreviation dictionaries *abbreviations.com* and *acronymfinder.com* are used for online lookup. The websites are

crawled using the Java HTML parser jsoup. The lists of meanings for an abbreviation are merged, and a *Disambiguation* is created and added to the persistent cache. Both sites allow user participation to add abbreviations to the site. The file-based approach was chosen to allow for reproducibility despite the volatility of the online sources. This allows the distribution of abbreviation cache files with the Benchmark projects.

The *AbbreviationDisambiguationHelper* can ambiguate a text using the known abbreviations from the cache using *ambiguateAll*. This functionality is used by the Similarity Metrics (SMs) as explained in Section 3.2. This is done by iterating over all disambiguations and replacing occurrences of the meanings in the text with abbreviations.

DBPediaHelper The *DBPediaHelper* implements the automatic creation of blacklists containing computer- and software-related terminology from the online ontology *DBpedia*. *DBPediaHelper* implements a *FileBasedCache*. The file contains a list of programming languages, markup languages, and software. Similarly to the *AbbreviationDisambiguationHelper*, this approach was chosen because it allows the distribution of blacklist files to ensure reproducibility. If no file is provided, a file is loaded from the results of SPARQL queries to the *DBpedia* SPARQL endpoint. The *DBpedia* ontology is a crowd-sourced cross-domain ontology that contains information from multiple encyclopedias, including Wikipedia. The format of entries is standardized, but the quality varies. Searching for entries with properties associated with a specific category can be used to mitigate the commonly observed problem of miscategorized information. An example is the query used to retrieve the set of programming language labels from *DBpedia* in Listing 4.2. It queries the programming language type derived from the *Yago* ontology and *DBpedia*'s own programming language type. The *DBpedia* programming language type contains many entries that are not programming languages. However, proper entries often have the fields *influenced* and *influencedBy*. For example, the entry for *Python* contains the programming languages *GO* and *Cobra* as values for *influenced*, and *ABC* as value *influencedBy*. Requiring the existence of such fields reduces the amount of unrelated noise retrieved by the query. The *DBPediaHelper* provides the three methods *isWordProgrammingLanguage*, *isWordMarkupLanguage*, and *isWordSoftware*. The methods require an exact case-insensitive match to prevent words from being filtered too aggressively.

DiagramUtil *DiagramUtil* is a utility class that encapsulates shared functionality that is related to diagrams and *DiagramElements*. *DiagramUtil* provides methods to calculate the similarity between a *Box DiagramElement* and a *RecommendedInstance* or *Box*. *CalculateSimilarityMap* calculates a map containing the highest similarity of each word from a *RecommendedInstance* and a *Box*. *CalculateHighestSimilarity(Word, Box)* calculates the similarity of the *Word* to all *Box* references and returns the maximum. The overload *CalculateHighestSimilarity(NounMapping, Box)* uses *CalculateHighestSimilarity(Word, Box)* to find the maximum similarity of any of its contained words to the box. The similarity functionality is primarily used to determine the confidence of *LinkBetweenDeAndRi* instances in the Diagram Connection Generator stage and in the *DiagramBackedTextStateStrategy*.

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo:<http://dbpedia.org/ontology/>
PREFIX yago:<http://dbpedia.org/class/yago/>

SELECT ?label
WHERE {
    {
        ?pl dbo:abstract ?abstract .
        ?pl rdfs:label ?label .
        ?pl rdf:type yago:ProgrammingLanguage106898352 .
        FILTER (LANG(?abstract) = 'en') .
        FILTER (LANG(?label)='en')
    }
    UNION
    {
        ?pl dbo:abstract ?abstract .
        ?pl rdfs:label ?label .
        ?pl dbo:influenced ?influenced .
        ?pl dbo:influencedBy ?influencedBy .
        ?pl rdf:type dbo:ProgrammingLanguage .
        FILTER (LANG(?abstract) = 'en') .
        FILTER (LANG(?label)='en')
    }
}
GROUP BY ?label
```

Listing 4.2: SPARQL query for a set of programming language labels

4.2 Diagram Gold Standard

The gold standard, as described in Section 5.3, is accessible by an implementation in the *test-resources* module. The following classes are used for loading the gold standard:

- *DiagramFromGoldStandard*
- *DiagramElement*
- *TextBox*
- *BoundingBox*

The *DiagramFromGoldStandard* class encapsulates the information related to a diagram and is an implementation of the *Diagram* interface provided by Linking Sketches and Software Architecture (LiSSA). The information consists of *DiagramElements*, *TextBoxes*, *DiaGSTraceLinks*, and parts of the diagram metadata (e.g., diagram resource name, diagram file handle, and name and file handle of the gold standard). *DiagramElement* is the superclass of all geometric shapes in a diagram. Each *DiagramElement* is identifiable by the *Diagram* it belongs to and its *BoundingBox*. A *DiagramElement* may contain an arbitrary amount of *TextBox* instances. The class also provides functionality for calculating subelements within its *BoundingBox*. *DiagramElement* currently has one subclass *Box*. The *Box* class provided by LiSSA represents a *DiagramElement* instance with a box shape. *TextBox* and *BoundingBox* already existed prior to this work. The *TextBox* class contains bounding box coordinates, text as a String, and a confidence value. The original *BoundingBox* record was extended with methods to calculate intersection, union, and intersection over union with another bounding box. These classes are sufficient for the manually extracted diagram element gold standard **GS3**. For the trace link gold standard **GS4**, the classes *DiaTexTraceLink*, *DiaGSTraceLink*, and *DiaWordTraceLink* were implemented. An excerpt of the implementation is depicted in Figure 4.2.

DiaTexTraceLink is the superclass of both *DiaGSTraceLink* and *DiaWordTraceLink*. A *DiaTexTraceLink* represents a trace link between a *DiagramElement* and a *Sentence*. *Sentences* are extracted from the text file associated with a project from the *Benchmark* as described in Section 2.5. The *DiaGSTraceLink* and *DiaWordTraceLink* subclasses were implemented to allow distinguishing between trace links generated from the gold standard and trace links generated from the approach, as well as to encapsulate additional properties that each may have depending on their origin. A *DiaGSTraceLink* stores a reference to the textual gold standard it was derived from. This is necessary because some benchmark projects contain multiple different text files. Architecture Documentation Consistency (ArDoCo) operates on a single text file basis. Therefore, it is necessary to filter the gold standard trace links according to the text input. The *DiaWordTraceLink* class was created to reflect that ArDoCo's TLR approach works on a word and phrase basis. This type of trace link additionally stores a reference to the *Word* instance it can be traced back to. This information is useful for debugging because it provides more comprehensible information about the origin of a trace link than a sentence with a potentially large number of irrelevant words. The information may also be used for heuristics in the future.

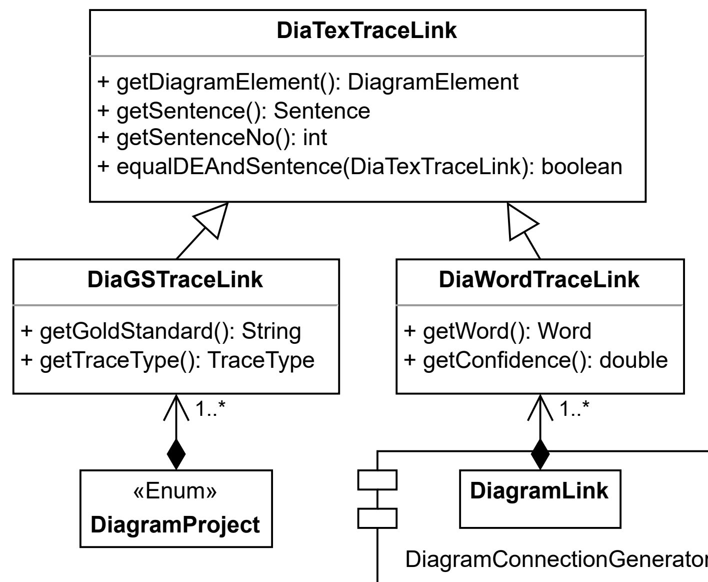


Figure 4.2: Trace Link classes excerpt in UML

The gold standard is used by various stages and should be usable regardless of any specific stage implementation. Therefore, parsing the JSON gold standard to create *Diagram* instances is delegated to the module containing the gold standard. ArDoCo encapsulates functionality to access the gold standard using the enum *Project*. *Project* enumerates all projects in the gold standard and contains functionality for accessing the associated files. The gold standard files were originally located in the *tests* module as test/resources. This was problematic for a variety of reasons:

- The *tests* module, as well as its submodules, were not deployed, preventing their usage outside of cloning the main repository
- Even if the modules were deployed, test/resources are not deployed by Maven, further preventing access
- The files were referred to using relative paths, preventing use outside of *tests* parent module

This issue was initially circumvented by moving the gold standard into a new *test-resources* module and moving the files to main/resources. The enum *DiagramProject* was created to enumerate all existing *Project* enum instances, which were extended with diagrams. The *Project* enum and various stages rely on file handles when accessing the gold standard. Therefore, the *DiagramProject* enum must also support file handles. However, since the module *test-resources* was supposed to mitigate the aforementioned issues, it needed to be deployable. After deploying as a JAR, accessing the resources as files is no longer possible. This issue was resolved by creating temporary files when accessing a gold standard resource. After consultation a solution based on this approach was implemented for ArDoCo, release 0.13.0 with the files now residing in the module *test-base* and the test

modules being deployed to the public maven repository. However, the *DiagramProject* enum and related classes still reside within *test-resources*.

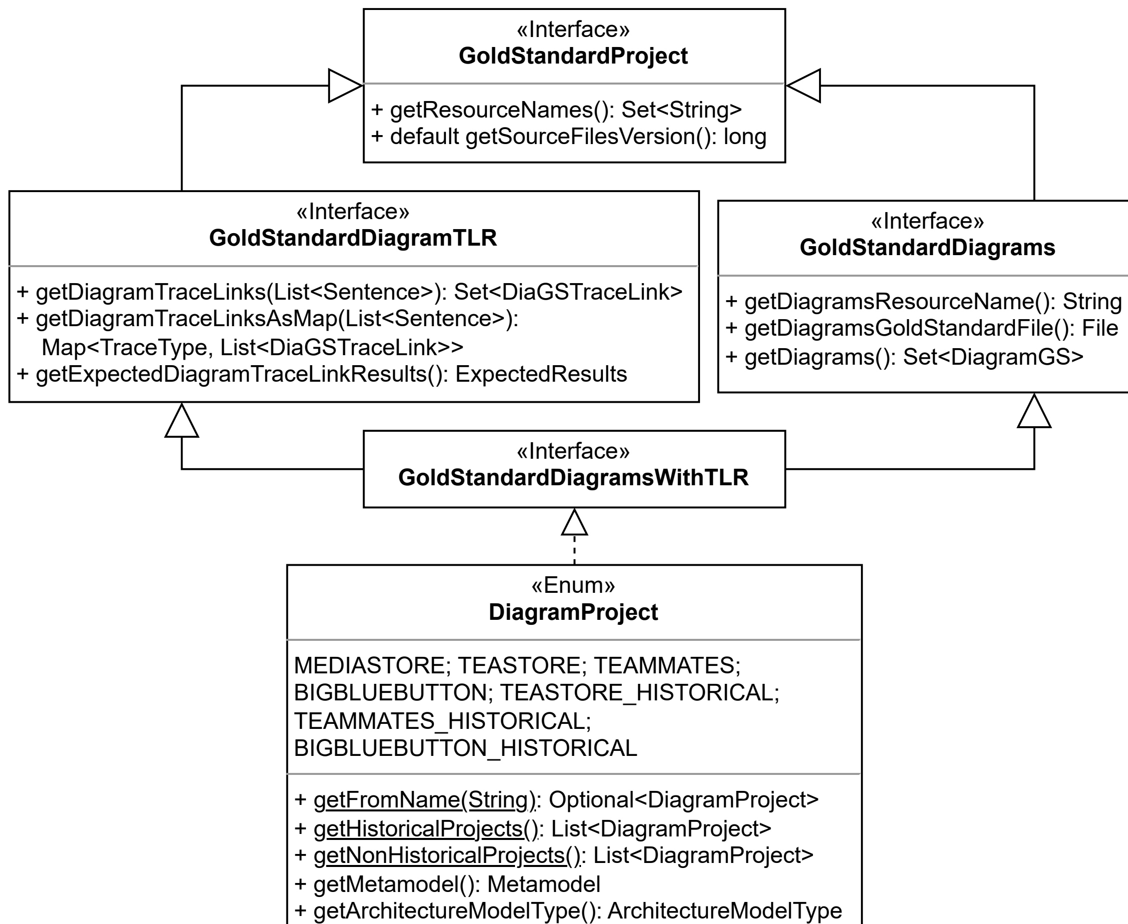


Figure 4.3: DiagramProject structure excerpt in UML

Because enums can not be extended in Java, a different structure is required to depict *DiagramProject*'s relation to *Project*. Figure 4.3 shows the structure's implementation. The functions defined in *Project* are moved to the *GoldStandardProject* interface with their default implementation residing in *Project*. However, these methods are not depicted in the figure to reduce clutter. This structure has the advantage that the interfaces can be referenced in declarations rather than referencing *Project* or any related enum directly. This provides more flexibility since interfaces can be extended and implemented by an arbitrary number of classes.

GoldStandardProject The *GoldStandardProject* interface is implemented by *Project* and all related gold standard enums. As previously explained, the default implementation of all functions refers to the *Project* enum. The interface provides functionality to access the name of each independent resource and file handles for each resource. *GetResourceNames* provides a list consisting of all the resource names associated with an instance. This function is required for the default implementation of *getSourceFilesVersion*. *getSourceFilesVersion* reads all resources associated with the instance and calculates a MD5

checksum for each. If the checksum deviates from the known checksum or no checksum was established previously, the source files version is updated. If caching is performed in a stage, this function may be used to invalidate caches if the source files changed.

GoldStandardDiagrams This interface provides functionality that reflects a gold standard containing diagram elements (The *DiagramElements* of a diagram can be accessed from the *DiagramFromGoldStandard* instance). *GetDiagramsResourceName* returns the resource name of the diagrams gold standard file. If a file handle is required instead, *getDiagramsGoldStandardFile* can be used. As previously explained, file handles can be problematic in the context of JAR packaging. However, the existing *Diagram* interface requires a file handle.

GoldStandardDiagramTLR The *GoldStandardDiagramTLR* interface contains functionality for retrieving *DiaGSTraceLink* trace links from the gold standard. The *getDiagramTraceLinks* method retrieves all trace links from the gold standard file. The trace links are initialized with their corresponding sentence during retrieval. *getDiagramTraceLinksAsMap* retrieves a map of trace links according to their *TraceType*.

GoldStandardDiagramsWithTLR This interface extends *GoldStandardDiagrams*, as well as *GoldStandardDiagramTLR*. Therefore, it represents a gold standard capable of retrieving *DiagramElements* and *DiaGSTraceLink* trace links. The approach of providing separate interfaces and a combined interface was chosen to avoid limiting the implementation to a shared *DiagramElement* and trace links gold standard. For example, this allows stages to specify that they only require a gold standard project with access to *DiagramElements*, but do not require the linkage information.

DiagramProject This gold standard implementation implements the *GoldStandardDiagramsWithTLR* interface backed by a JSON file containing the *DiagramsFromGoldStandard*, *DiagramElements*, *TextBoxes*, *BoundingBoxes*, and *DiaGSTraceLinks*. The static *getFromName* method retrieves an *Optional* containing the *DiagramProject* instance with the given name or an empty *Optional*. This can be useful because ArDoCo only stores project names during execution. *Get(Non)HistoricalProjects* returns the list of (non-)historical enum instances, which is useful for parameterized testing. *GetMetamodel* returns the *Metamodel* enum instance of the project. The *Metamodel* determines whether the project provides an architecture model or code model. The *getArchitectureModelType* method returns the *ArchitectureModelType* enum instance of the project, which determines whether a Palladio Component Model (PCM) or Unified Modelling Language (UML) model is provided by the source files. The gold standard is parsed from JSON using Jackson[1]. It is parsed using the following classes:

- *DiagramsFromGoldStandard* - Contains an array of *DiagramFromGoldStandard*, as well as the name of the JSON schema used by the gold standard.
- *DiagramFromGoldStandard* - Implements the LiSSA *Diagram* interface and is responsible for parsing the boxes contained in the diagram.
- *BoxFromGoldStandard* - Extends the existing *Box* implementation of LiSSA. It keeps a list of sub-boxes that corresponds to the nesting of boxes in the JSON file and a set of *TraceLinkFromGoldStandard*.

[1] See FasterXML/jackson

- *TraceLinkFromGoldStandard* - Record representing the trace links associated with the parent box. It contains the name of the text gold standard it references and the sentence number of each trace link. Additionally, it contains a potentially empty set of *TypedTraceLinkFromGoldStandard*. The *toTraceLinks* function creates *DiaGSTraceLinks* from the data.

- *TypedTraceLinkFromGoldStandard* - A set of trace links with a specific *TraceType*. Depending on the *TraceType*, the specified links do not necessarily constitute True Positives (TPs) but can be used for debugging purposes to identify potential sources of False Positives (FPs) and False Negatives (FNs). They are automatically converted by their *TraceLinkGS* parent, if *toTraceLinks* is called and added to the collection of *DiaGSTraceLinks*.

- *TraceType* - See Section 5.3 for a detailed explanation of possible values.

- *BoundingBoxDeserializer & TextBoxDeserializer* - Custom Jackson deserializer for LiSSA's *BoundingBox* and *TextBox*.

4.3 Diagram Recognition Module

This section is about the adjustments and extensions to the implementation in the *diagram-recognition* module. The module contains the *Diagram Recognition* stage, which was implemented as part of LiSSA and a new *Diagram Recognition Mock* stage. The changes and extensions to the *Diagram Recognition* stage are detailed in Subsection 4.3.1. The implementation of the *Diagram Recognition Mock* stage is depicted in Subsection 4.3.2. Subsection 4.3.3 details the implementation of a *TextStateStrategy*, which uses the *DiagramElements* in its word clustering process.

4.3.1 Diagram Recognition

The *Diagram Recognition* stage is responsible for transforming the pictures of informal diagrams into a representation that can be used programmatically. The stage and corresponding module *diagram-recognition* was created as part of LiSSA. This work extends the stage with two additional agents, two informants, caching, and improvements to the *TextBox* assignment. The module now also contains the *DiagramBackedTextStateStrategy*, which can be used by the *Text Extraction* stage. As described in Section 3.3, diagram-related stages should positively influence the similarity of homoglyphs and confusables. To achieve this, a custom *Character Match Function* can be set in ArDoCo's *WordSimUtils* class, which determines whether two characters are considered a match. Because of this, the *Diagram Recognition* stage and all other diagram-related stages are adjusted to temporarily change the *Character Match Function* to a function that also matches homoglyphs and confusables. The implementation of this function is detailed in Section 4.6.

DiagramRecognitionAgent Adjustments The *Diagram Recognition* stage uses Object Detection (OD) and Optical Character Recognition (OCR) processes in an external

docker container. These processes take up most of the time required for a run and must run for each project. To avoid performing the same OD and OCR process every time ArDoCo runs a Benchmark project, a cache is added to the Kotlin *DiagramRecognitionAgent* agent. This cache is implemented as a *SerializableFileBasedCache*. A cache file containing a *SketchRecognitionResult* is created for each diagram picture. If the diagram was processed in a previous run, the *SketchRecognitionResult* is deserialized, and the diagram is skipped. Otherwise, the diagram is processed, and the *SketchRecognitionResult* is serialized and cached for the subsequent runs.

The *DiagramRecognitionAgent* agent also struggled with *TextBoxes* in nested *DiagramElements*. This is because the image processing creates *DiagramElements* and *TextBoxes* separately at first. Afterwards, the *TextBoxes* are combined and assigned to a *DiagramElement* based on the *Intersection over Union* metric. For nested *DiagramElements*, the assignment breaks because a *TextBox* may be contained within the boundary box of multiple *DiagramElements*. To fix this, the *DiagramElements* now implement a parent-child relationship, where the parent is calculated by finding the smallest *DiagramElement*, which entirely contains the child's bounding box. The *DiagramRecognitionAgent* implementation is adjusted to utilize the parent-child relationship by performing a depth-first search approach. If the *TextBox* is assigned to a child, it can no longer be assigned to a parent. This solves the issue above.

DiagramDisambiguationInformant The *DiagramDisambiguationInformant* informant iterates over all *TextBoxes* in the diagrams. The informant uses the *getAbbreviationCandidates* function of the *AbbreviationDisambiguationHelper* utility class to determine the set of possible abbreviations in the *TextBox*. Afterwards, it uses the *disambiguate* function of the utility class to try and disambiguate each candidate. The meanings are retrieved from the cache if the candidate is a known abbreviation. Otherwise, an online lookup is performed as described in Section 4.1. A new *Disambiguation* instance is created for each resolved abbreviation and added to the *DiagramRecognitionState*. The informant is executed by the *DiagramDisambiguationAgent*. The agent is executed penultimate, followed by the *DiagramReferenceAgent*.

DiagramModelReferenceInformant The *DiagramModelReferenceInformant* is responsible for calculating references for *DiagramElements*. References are used to compare *DiagramElements* to text. The *Diagram Connection Generator* utilizes this to perform a similarity-based Traceability Link Recovery (TLR). The *DiagramModelReferenceInformant* is called by the *DiagramReferenceAgent*. The agent and informant are executed after all other agents to ensure that the *DiagramElements* are final before references are calculated. The references are calculated for each *DiagramElement* of each diagram.

The informant first calculates the references for each *TextBox*. The text of a box is split at brackets and enumerations (e.g. "Lorem(ipsum),dolorSit"→"Lorem","ipsum","dolorSit"). Afterwards, *CamelCase* is split up to produce split and decameled references (e.g. "dolorSit"→"dolor","Sit"). Both splits are performed using regular expressions. The results are combined and filtered using the *DBPediaHelper* to remove computer- and software-related terminology. An additional reference is added by joining the split and decameled references. This implementation intentionally creates many references because they are

filtered afterwards. After the *TextBox* references are calculated, the first filter is applied. This filter checks whether references with uppercase characters exist. If this is the case, the lowercase references are discarded. Otherwise, the lowercase references are preserved.

After all *TextBox* references have been calculated, the *DiagramModelReferenceInformant* calculates the *TextBox* that is most similar to a model instance. This is done because a *DiagramElement* may directly correspond to a model instance. If this is the case, the proper reference of the *DiagramElement* is assumed to be similar to the model instance. In this case, the other *TextBoxes* are considered to be supplemental. Therefore, if the similarity between a *TextBox* and a *DiagramElement* is above a configurable threshold, all references from other *TextBoxes* are discarded.

4.3.2 Diagram Recognition Mock

Because the first implementation phase focuses on the TLR between manually extracted *DiagramElements*, a stage is required that provides these elements. The purpose of this stage consists of retrieving the *Diagrams* from the gold standard and subsequently making them accessible to later pipeline stages. To ensure a seamless transition between manual and automatic extraction in later project stages, the stage is implemented to replace the *Diagram Recognition* stage. The LiSSA *Diagram Recognition* stage is responsible for automatic diagram recognition and was already implemented in Kotlin before this work. It provides a *DiagramRecognitionState* data structure. The data structure offers access to the set of extracted *Diagrams* and the *DiagramElements*, *TextBoxes*, and *BoundingBoxes* they consist of. The *Diagram Recognition Mock* stage uses the same data structure as output. To comply with ArDoCo's existing project structure, the *Diagram Recognition Mock* stage resides in the maven module *diagram-recognition* as a submodule of the *stages* parent module. Because the gold standard is independent of a specific stage implementation, parsing the raw JSON files is delegated to the gold standard implementation. The *Diagram Recognition Mock* receives a *GoldStandardDiagramsWithTLR* gold standard project instance as input. This gold standard project instance can parse the gold standard described in Section 4.2. The stage's task consists of saving the parsed *Diagrams* to the *DiagramRecognitionState* data structure during state initialization. The stage can be run independently and has no other pipeline data structures as a prerequisite.

4.3.3 Diagram-Backed Text State Strategy

The *DiagramBackedTextStateStrategy* extends ArDoCo's *OriginalTextStateStrategy*. A *TextStateStrategy* is responsible for the creation and merging of *NounMapping* instances. The strategy used by the *Text Extraction* can be set before executing the *Text Extraction* stage. This allows changing the composition of *NounMapping* instances without modifying the *Text Extraction* and its agents directly. The *DiagramBackedTextStateStrategy* differs from the original strategy by incorporating the relation between *Words* and *DiagramElements* into the *NounMapping*. The *NounMappingImpl* is extended by the *DiagramBackedNounMappingImpl* class. A *DiagramBackedNounMappingImpl* has the same functionality as the original implementation but keeps a reference to a *DiagramElement*. The *DiagramBackedTextStateStrategy* creates these mappings by searching for the *DiagramElement* with the

highest similarity to the *Words* of the mapping. The *calculateHighestSimilarity* function of *DiagramUtil* is used to calculate the similarity of the mapping with each *DiagramElement*. If it surpasses a configurable threshold, the *DiagramElement* with the highest similarity is chosen. The original strategy tries to find a similar mapping before adding a *NounMapping* to the text state. If such a mapping is found, the mappings are merged. In contrast, the *DiagramBackedTextStateStrategy* merges the mappings if they are similar and backed by the same *DiagramElement*. This prevents falsely clustering *Words* that are unlikely to belong to the same entity.

4.4 Diagram Connection Generator

The *Diagram Connection Generator* stage implements the trace link creation between *DiagramElement* instances and *RecommendedInstances*. The stage is implemented by the *DiagramConnectionGenerator* class in the *stages* submodule *diagram-connection-generator*. The stage uses a single agent that is explained after the states.

The stage is made up of the two data structures *DiagramConnectionStates* and *DiagramConnectionState*. The *DiagramConnectionStates* data structure contains a *DiagramConnectionState* for each *Metamodel*. This is necessary because different *RecommendedInstances* are extracted for different metamodels. The *DiagramConnectionState* stores *LinkBetweenDeAndRi* instances. The *LinkBetweenDeAndRi* class represents a trace link between a *DiagramElement* and a *RecommendedInstance*. The confidence that the link is correct is determined using the confidence map, which contains confidence values for each *Word* associated with the *RecommendedInstance*. To reduce this map to a singular confidence value, *LinkBetweenDeAndRi* provides multiple static mapping lambda functions, which can be passed as a parameter to the *getConfidence* method. Maximum, minimum, average, and median are currently available for this purpose. The class also provides *toTraceLinks* to convert a *LinkBetweenDeAndRi* to a set of *DiaWordTraceLink* instances. This is achieved by creating a *DiaWordTraceLink* for each entry of the confidence map. Conversions are performed on demand because *DiaWordTraceLink* is only used for evaluation purposes in this work. The *DiagramConnectionState* offers the following four conversion methods:

- getWordTraceLinks(): Set<DiaWordTraceLink> - Converts all *LinkBetweenDeAndRi* instances and joins the results. Removes low confidence *DiaWordTraceLink* instances according to a configurable threshold.

- getTraceLinks(): Set<DiaWordTraceLink> - Set of trace links reduced to unique *DiagramElement* and *Sentence* combinations based on *getWordTraceLinks*.

- getMostSpecificWordTraceLinks(): Set<DiaWordTraceLink> - Set of trace links reduced to highest confidence *Diagram* and *Word* combinations based on *getWordTraceLinks*. If multiple links share the highest confidence for a combination, all of them are retained.

- getMostSpecificTraceLinks(): Set<DiaWordTraceLink> - Set of trace links reduced to the highest confidence *Diagram*, *DiagramElement* and *Word* combinations based on

getMostSpecificWordTraceLinks. Guarantees that each combination of *DiagramElement* and *Sentence* is unique.

GetMostSpecificTraceLinks is the function used to retrieve the *Predicted Positives* for evaluation. It reduces duplicate trace links pointing to the same *Word* due to ambiguous *DiagramElements* names. This is especially relevant for package diagrams because the names of packages and sub-packages deviate only slightly.

The *DiagramConnectionGenerator* implements the *DiagramConnectionAgent*, which is responsible for the creation of *LinkBetweenDeAndRi* instances. The *DiagramConnectionAgent* manages the creation, enabling, and disabling of informants, but apart from that, it contains no logic by itself. The agent uses the informants *DiagramAsModelInformant*, *DiagramTextInformant* and *LinkBetweenDeAndRiProbabilityFilter*, which are explained in the following paragraphs.

DiagramAsModelInformant The *DiagramAsModelInformant* creates temporary *ModelInstances* for each *DiagramElement.* This allows using ArDoCo's TLR process to find trace links between the *RecommendedInstances* and the temporary *ModelInstances.* For each trace link between a *RecommendedInstance* and a temporary *ModelInstance*, a *LinkBetweenDeAndRi* is created for the original *DiagramElement* and the *RecommendedInstance.* The conversion is performed by creating a new *ModelInstanceImpl* for each *DiagramElement* reference with the reference as a name.

DiagramTextInformant This informant iterates over all *TextBoxes* and *RecommendedInstances.* It aims to find initialisms in the *DiagramElements*, which can be resolved to the name of a *RecommendedInstance.* For each *TextBox*, the text is compared to the name of the *RecommendedInstance* using the *isInitialismOf* function from the *AbbreviationDisambiguationHelper.*

LinkBetweenDeAndRiProbabilityFilter The *LinkBetweenDeAndRiProbabilityFilter* iterates over all *LinkBetweenDeAndRi* instances that were found. For each link, the confidence is calculated from the confidence map using an aggregation function that calculates the maximum value contained in the map. The link is removed from the state if this confidence is below a configurable threshold. This filter is very forgiving because it promotes a higher recall. This is important to prevent the approach from falsely disaffirming *RecommendedInstances* in the later stages.

4.5 Diagram Inconsistency Checker

The *Diagram Inconsistency Checker* stage is responsible for finding Missing Diagram Element (MDE) and Unmentioned Diagram Element (UDE) *Inconsistencies* between the *DiagramElements* and *RecommendedInstances.* It is implemented in the *diagram-inconsistency-checker* submodule of the *stages* module in the *DiagramInconsistencyChecker* class. The stage has the two agents *DiagramInconsistencyAgent* and *RecommendedInstancesConfidenceAgent*, which are executed in this order and explained after the states below.

The stage contains the states *DiagramInconsistencyStates* and *DiagramInconsistencyState*. Similarly to the previously explained *DiagramConnectionStates* state, the state is used to provide a *DiagramInconsistencyState* for each *Metamodel*. The *DiagramInconsistencyState* stores a set of *Inconsistencies* that can be extended with *addInconsistency*. To retrieve the set of *Inconsistencies*, the *getInconsistencies* functionality is provided. The function allows specifying which type of *Inconsistency* should be returned.

The MDE and UDE *Inconsistencies* are implemented as *MDEInconsistency* and *UDEInconsistency*. These classes implement the *Inconsistency* interface that ArDoCo uses for the Unmentioned Model Element (UME) and Missing Model Element (MME) *Inconsistencies*. The *Inconsistencies* are implemented as Java records. A *MDEInconsistency* record is a wrapper for a single *RecommendedInstance*, which does not have a corresponding *DiagramElement*. In contrast, a *UDEInconsistency* record is a wrapper for a single *DiagramElement*, which does not have a corresponding *RecommendedInstance*. Apart from basic methods that allow printing the *Inconsistency* record to a text file, the record contains no logic.

The states are populated by *DiagramInconsistencyAgent*. This agent is responsible for creating *MDEInconsistency* and *UDEInconsistency* instances and adding them to the *DiagramInconsistencyState*. To do this, the informants *MDEInconsistencyInformant* and *UDEInconsistencyInformant* are implemented. After the *DiagramInconsistencyAgent* is finished, the *RecommendedInstancesConfidenceAgent* is executed to change the confidence of each *RecommendedInstance* affected by an *Inconsistency*. The *InfluenceByInconsistenciesInformant* is implemented to achieve this.

MDEInconsistencyInformant This informant creates *MDEInconsistency* instances. The informant iterates over all available *Metamodels* and gets the corresponding *RecommendationState*, *DiagramConnectionState* and *DiagramInconsistencyState*. It then retrieves all *RecommendedInstances* from the *RecommendationState* and all *LinkBetweenDeAndRi* instances from the *DiagramConnectionState*. Afterwards, the informant determines all *RecommendedInstances* which are not endpoints of a *LinkBetweenDeAndRi* instance. Each of these *RecommendedInstances* is wrapped in a *MDEInconsistency* instance and added to the*DiagramRecognitionState*.

UDEInconsistencyInformant The *UDEInconsistencyInformant* is structured like the *MDEInconsistencyInformant*, but instead of getting the *RecommendationState*, it gets the *DiagramRecognitionState* from the *DataRepository*. The informant retrieves all *DiagramElements* from the *DiagramRecognitionState*. Afterwards, the informants determine all *DiagramElements* which are not an endpoint of a *LinkBetweenDeAndRi* instance. A *UDEInconsistency* instance is created and added to the *DiagramRecognitionState* for each.

InfluenceByInconsistenciesInformant This informant influences the confidence in each *RecommendedInstance* affected by an *Inconsistency*. Like the previous informants, the informant iterates over all available *Metamodels* and retrieves the corresponding *RecommendedInstances*, *LinksBetweenDeAndRi*, and *Inconsistencies* from the states. To comply with heuristic **D1** in Section 3.3, the informant calculates what percentage of *RecommendedInstances* is contained in a *LinkBetweenDeAndRi* instance. The coverage is calculated by dividing this number by the amount of total *RecommendedInstances*.

Afterwards, the function *punishInconsistency* is used to reduce the confidence of *RecommendedInstances* affected by an *MDEInconsistency*. The confidence reduction depends on a configurable maximum punishment, the coverage, and the number of *Claimants*. In ArDoCo, a *Claimant* class claims a result. Each *RecommendedInstance* has a set of *Claimants* that claim its existence with a certain confidence. ArDoCo aggregates the confidences to give the overall confidence in a *RecommendedInstance*. For a higher amount of *Claimants*, the amount of punishment by the informant should be less severe. This prevents the *InfluenceByInconsistenciesInformant* from overruling an unreasonable amount of other *Claimants*. To determine the confidence punishment, the coverage is used to scale the configurable maximum punishment. This is proportional based on the assumption that a higher coverage indicates that a *MDEInconsistency* should be less likely.

The *rewardConsistency* function operates similarly to the *punishInconsistency* function. Each *RecommendedInstance* covered by a *LinkBetweenDeAndRi* gets a confidence reward based on a configurable maximum reward, the coverage and amount of *Claimants*. However, in this case, the coverage is inversely proportional to the reward. If only a few *RecommendedInstances* are covered, the informant assumes that the covered instances are particularly important and should be rewarded more.

4.6 Homoglyphs and Confusables

The text file "confusablesSummary" containing the Unicode confusables is publicly available and was chosen as a basis for implementing the *ConfusablesHelper* class. To use the confusables, the file had to be parsed and saved in a data structure. Because the confusables impact the similarity calculation in multiple stages, a global utility class *ConfusablesHelper* was implemented in the *common* module. The *ConfusablesHelper* loads the "confusablesSummary" file from resources and parses it line-by-line. The file consists of blocks of confusables as depicted in Figure 4.4. The first row of the block contains the sequence of confusable characters and always begins with a hash symbol. After the initial row, a row for each confusable is appended. These rows contain the symbol, its code point in hexadecimal notation, and a descriptive name of the character. Each block is surrounded by a single new line and adheres to the same structure. The confusable Unicode characters are extracted from the first row and added to a map, where each key contains a list of confusables and itself. This implementation uses a key-value pair for every *UnicodeCharacter* of each line, which requires more memory than only keeping a single list for each set of confusable *UnicodeCharacter* instances. However, performance was the primary concern in this case because the list of *UnicodeCharacter* instances need to be retrieved on a per-character basis, and this allows accessing the list of confusables for a character in $O(1)$, rather than having to search all lists to find which one contains the character.

A confusable impacts the Similarity Metrics (SMs) by causing a mismatch for characters that should match in a visual context. Therefore, the approach of including knowledge about confusables in the calculation performed by the character-based SMs was chosen. This was achieved by forcing the SMs to use a *Character Match Function* rather than comparing characters directly. A *Character Match Function* compares two characters and

```
#       !       !       !         !
        ( ! )   0021      EXCLAMATION MARK
←       ( ! )   01C3      LATIN LETTER RETROFLEX CLICK
←       ( ! )   2D51      TIFINAGH LETTER TUAREG YANG
←       ( ！ )  FF01      FULLWIDTH EXCLAMATION MARK      # →!→
```

Figure 4.4: A block of confusables from the confusablesSummary file

returns a boolean value to indicate whether they are considered a match. The default *Character Match Function UnicodeCharacter.EQUAL* is set in ArDoCo's *WordSimUtils* class and performs an equality check. *WordSimUtils* can also be set to an optional *Character Match Function EQUAL_OR_HOMOGLYPH*, which considers two characters a match if they are equal or homoglyphs according to the *ConfusablesHelper*. The benefit of this implementation is that it is configurable because stages can set the appropriate function before stage execution and reset it after stage execution. Therefore, this feature is isolated from stages, which do not require homoglyphs to be considered equal.

During implementation, an existing design flaw in the implementation of the character-based SMs was discovered. Java natively provides the *String* and *Character* class, as well as the corresponding primitive *char* for text. However, these classes were initially created before the Unicode extension beyond 16-bit[2]. A *Character* can only cover the *Basic Multilingual Plane* from U+0000 to U+FFFF. The Unicode characters above U+FFFF are referred to as supplementary characters and can not be represented by a single *Character*. This can lead to unintuitive behavior for basic text operations relying on *Characters* internally. For example, the *String* **u** has length two and consists of two separate characters. *String* operations such as *substring*, *split*, et cetera do not consider this and can break Unicode characters apart. This issue also affects the *ConfusablesHelper* class because many confusables are not part of the *Basic Multilingual Plane*. To handle Unicode inputs, the classes *UnicodeCharacter* and *UnicodeCharacterSequence* were implemented to replace *String* and *Character (char)* where suitable. A *UnicodeCharacter* consists of an *Integer* representing the code point in the Unicode code space and a *String* containing the representation. The class also provides functionality for conversion between *Strings* and *UnicodeCharacter* instances and includes the *Character Match Functions*. The *UnicodeCharacter* class is designed to cover the entire Unicode code space. The problems concerning *Strings* are based on the fact that *Strings* are implementations of *CharSequence* in Java. Therefore, a corresponding *UnicodeCharacterSequence* was implemented. The *UnicodeCharacterSequence* provides the same functionality as *CharSequence* but relies on *UnicodeCharacter* instances rather than Java's *char* primitive. The class also provides conversion functionality for conversion between *Strings* and the corresponding *UnicodeCharacterSequence*.

ArDoCo currently uses the character-based *Jaro-Winkler* and *Levenshtein* SMs. Before this work, both SMs were imported from an external Apache Commons module. In compliance with their licensing, the source code of the SMs was moved to the *common*

[2] See Oracle's Supplementary Characters in the Java Platform

module and adapted to use the *UnicodeCharacter*, *UnicodeCharacterSequence* classes, as well as the selected *Character Match Function*.

5 Evaluation

This chapter contains the evaluation of Entity Recognition in Software Documentation Using Trace Links to Informal Diagrams (ERID). The evaluation aims to evaluate goals and questions established using the Goal Question Metric approach [2]. The initial Goal (G1) is to create trace links between textual entities and diagram elements. The second goal (G2) of this work is to find Unmentioned Diagram Element (UDE) and Missing Diagram Element (MDE) inconsistencies. The third goal (G3) and fourth goal (G4) are about the integration into Architecture Documentation Consistency (ArDoCo). G3 is to create trace links between textual entities and diagram elements extracted using Optical Character Recognition (OCR) and Object Detection (OD). G4 is to reduce the falsely identified textual entities. The goals are examined by answering the following collection of questions:

Q1 How well can the approach detect trace links between textual entities and manually extracted diagram elements?

Q2 How well can the approach detect trace links between textual entities and diagram elements extracted using OCR and OD?

Q3 To what degree does ERID affect ArDoCo's performance?

The following sections present the evaluation of the approach. The evaluation metrics are detailed in Section 5.1. The Diagram Text Traceability Link Recovery (TLR) is evaluated in regard to sentence numbers rather than textual entities. This requires a conversion between the *LinkBetweenDeAndRi* instances to a representation that can be used for the evaluation. This is further explained in Section 5.2. Section 5.3 explains how the two additional gold standards for the evaluation were created. These gold standards are used in Section 5.4 to evaluate the Diagram Text TLR. The impact on ArDoCo's overall performance is evaluated in Section 5.5. Lastly, Section 5.6 explains the threats to validity.

5.1 Metrics

The evaluation uses the same metrics that were used in prior work about ArDoCo by Keim et al. [17] to allow comparison to their previous results. This is especially relevant for question **Q3** because ArDoCo's performance was evaluated using these metrics in the paper. The metrics are Precision (P), Recall (R), F_1-score, Accuracy (Acc), Specificity (Spec) and the normalized Φ coefficient Φ_N. These metrics are defined in reference to True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN).

The evaluation consists of multiple binary classification problems. A binary classification is the most basic classification. It is the task of classifying an input into one of two classes:

Positive and *Negative* [22]. For example, a (potentially imperfect) medical test for a specific illness may classify a test subject as infected (*Positive*) or not infected (*Negative*). This test is a binary classifier, and the condition that represents each class is arbitrary but fixed. Assume that an oracle exists, which is a perfect binary classifier. When comparing a binary classifier to an oracle, the classification of an input by the binary classifier is referred to as *Predicted Outcome*, and the classification of the oracle is referred to as *Actual Value*. The *Positive* and *Negative* classes of the binary classifier are referred to as Predicted Positives (PPs) and Predicted Negatives (PNs) in this context. The oracle's classes are referred to as (Actual) Positives (APs) and (Actual) Negatives (ANs). The intersection of these classes is represented by True Positives (TPs), False Positives (FPs), True Negatives (TNs), and False Negatives (FNs). This is commonly visualized in a $2x2$ confusion matrix [22]:

Predicted Outcome / Actual Value	AP	AN
PP	TP	FP
PN	FN	TN

Figure 5.1: $2x2$ Confusion Matrix

The results can be used to determine the classifier's performance with the performance metrics. To determine the confusion matrix, a classifier is tasked with classifying inputs with known classes. In the context of this evaluation, these inputs are derived from the gold standards. The classes are introduced for each classification problem in the following (sub-)sections. To determine the values of TP, FP, TN, and FN, the following formulas can be used by interpreting each class as a set:

$$
\begin{aligned}
TP &= |PP \cap AP| \\
FP &= |PP - AP| \\
FN &= |AP - PP| \\
TN &= Total - (TP + FP + FN)
\end{aligned}
$$

where:

$Total$ is the amount of classified inputs

Equation 5.2: Formulas for TP, FP, TN, and FN

The equations in 5.2 have the advantage that neither the Predicted Negatives (PNs) nor Actual Negatives (ANs) are needed in the calculations. The evaluation relies on metrics commonly used in TLR research [17]. The metrics are listed and explained below.

Precision (P) Precision is the fraction of true positive predictions to all positive predictions [5]. It gives information about what percentage of reported positives is correctly classified. Precision ranges from 0, indicating that all positive predictions are false, to 1, indicating all positive predictions are true. Precision can not be assessed without positive predictions. A high Precision indicates that any given positive prediction is likely to be true. The Precision can be calculated using:

$$P = \frac{\text{true positive predictions}}{\text{all positive predictions}} = \frac{TP}{TP + FP}$$

Recall (R) Recall, or *True Positive Rate*, is the fraction of true positive predictions to all actual positives in a classification [5]. It gives information about what percentage of actual positives is correctly classified. Recall ranges from 0, indicating that no positive element was predicted as positive, to 1, indicating that all positive elements were predicted as positive. Recall can not be assessed if no positive elements exist. A high Recall indicates that any given positive element is likely to be predicted as positive. The Recall can be calculated using:

$$R = \frac{\text{true positive predictions}}{\text{actual positives}} = \frac{TP}{TP + FN}$$

F_1-score The F_1-score is used to combine the metrics P and R using the harmonic mean [37]. The F_1-score is a member of the family of metrics defined by the F_α-score, which weighs R α times as much as P. For $\alpha = 1$, the F_1-score weighs R and P equally. The F_1-score ranges from 0 to 1, with higher numbers indicating better. It can be calculated using:

$$F_\alpha = (1 + \alpha) * \frac{P * R}{\alpha * P + R}$$
$$F_1 = 2 * \frac{P * R}{P + R}$$

Accuracy (Acc) Accuracy is the ratio of correct predictions to total predictions. The metric is essential in defining how much confidence can be placed in a set of predictions. It is a combination of Precision and the *Trueness* (Bias), which relates to the non-random systematic error of the classification [29]. The metric ranges from 0 to 1, with 0 indicating that all predictions are false predictions and 1 indicating that all predictions are true predictions. It can be calculated using the formula:

$$Acc = \frac{\text{true predictions}}{\text{all predictions}} = \frac{TP + TN}{TP + FN + FP + TN}$$

Specificity (Spec) The Specificity, or *True Negative Rate*, is the fraction of true negative predictions to all actual negatives in a classification [17]. Therefore, Specificity is the counterpart to Recall. It ranges from 0 to 1. A value of 0 indicates that all negatives are falsely predicted as positive. A value of 1 indicates that all negatives are correctly predicted as negative. The Specificity is calculated using:

$$Spec = \frac{\text{true negative predictions}}{\text{actual negatives}} = \frac{TN}{TN + FP}$$

Φ coefficient The Φ coefficient, also referred to as Matthews correlation coefficient [27], measures how the predictions of a binary classification relate to the observed values. The Φ coefficient ranges from values of -1 to +1. According to Matthews, a value of 0 is comparable to random predictions. A value of +1 indicates perfect agreement between the predictions and the observation, with -1 indicating total disagreement [27]. However, these values can usually not be obtained [6]. Values of unity can be achieved by calculating the maximum value Φ_{max} and applying it to Φ. The Φ and the normalized Φ_N are calculated using the following equations, which are adapted from a paper by Keim et al. [17]:

$$\begin{aligned} PP &= TP + FP & PN &= TN + FN & R_1 &= \sqrt{PP * AN} \\ AP &= TP + FN & AN &= TN + FP & R_2 &= \sqrt{AP * PN} \end{aligned}$$

$$\begin{aligned} \Phi &= \frac{TP * TN - FP * FN}{R_1 * R_2} \\ \Phi_{max} &= \begin{cases} R_1/R_2 & AP \geq PP \\ R_2/R_1 & otherwise \end{cases} \\ \Phi_N &= \Phi/\Phi_{max} \end{aligned}$$

5.2 Diagram-to-Sentence Trace Links

A definition of a trace link is required to answer the evaluation questions and create the gold standard. In the context of this thesis, the trace links used for evaluation consist of a linkage between a sentence contained in the Natural Language Software Architecture Documentation (NLSAD) and a diagram element from an informal diagram. The internal representation of the trace links is called *DiaTexTraceLink* and is introduced in Section 4.2. This representation is analogous in structure to the *SadSamTraceLink* that exists in ArDoCo for the evaluation of its Software Architecture Documentation (SAD) to Software Architecture Model (SAM) TLR capabilities.

Each trace link recovered by the approach consists of a linkage between exactly one sentence and one diagram element as shown in Figure 5.3. The trace link represents an explicit mention of a diagram element in a sentence. For the trace links created by the approach, a trace link is tied to the existence of a *RecommendedInstance* that contains the

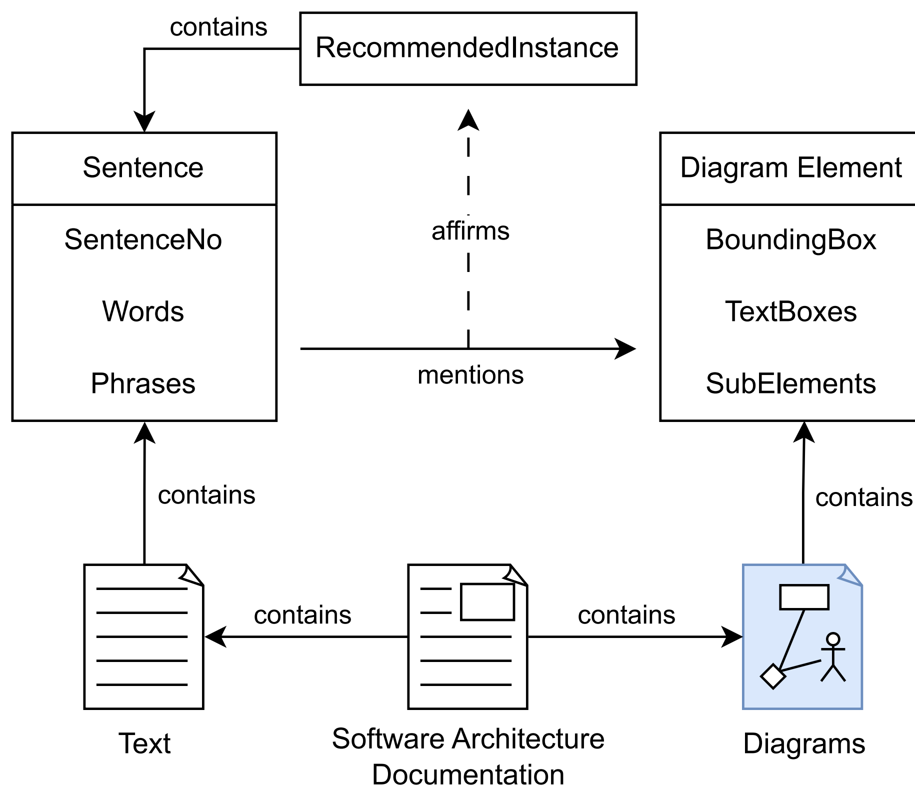


Figure 5.3: Diagram-to-sentence Trace Links

sentence. As part of the approach, the existence (or absence) of a trace link increases (or decreases) the confidence in the *RecommendedInstance* in later stages of ArDoCo. Initially, coarse *LinkBetweenDeAndRi* trace links are constructed between *RecommendedInstances* and diagram elements as described in Section 4.4. Diagram-to-sentence trace links are subsequently extracted from the created *LinkBetweenDeAndRi* by creating a diagram-to-sentence trace link for each sentence covered by the *RecommendedInstance* endpoint.

For the diagram-to-sentence trace links in the gold standard, the existence of a *RecommendedInstance* is not required. They only need the information about which sentence and diagram element they link. Because every project has one text file, a sentence number is enough to identify the sentence. To identify the diagram element, its bounding box coordinates and the name of the diagram resource are sufficient.

Gold standard and approach diagram-to-sentence trace links are compared by their sentence numbers and bounding boxes. The OD provides the bounding boxes of diagram elements from the approach. However, due to imprecision, these bounding boxes do not exactly match the bounding boxes from gold standard trace links. To determine if two diagram elements are the same, the metric *Intersection over Union* is used. The metric is calculated by dividing the area of the intersection by the area of the union [35]. The evaluation assumes an *Intersection over Union* score above 0.7 for bounding boxes to be considered matches.

5.3 Gold Standard Creation

As explained in the previous section, the possible endpoints must be known to evaluate a set of diagram-to-sentence trace links. Additionally, a list of diagram-to-sentence trace links between endpoints needs to exist. The endpoints, as previously described in Figure 5.3, require a list of sentences and a list of diagram elements. This thesis relies on and extends the ArDoCo *Benchmark* data set [11], which includes the sentences of all projects. Therefore, only the two additional gold standards **GS3** and **GS4** for the diagram elements and the links themselves were created and added to the ArDoCo *Benchmark* data set:

GS1 Set of expected trace links between Sentences and Model Elements

GS2 Set of expected Unmentioned Model Elements

GS3 Set of manually extracted Diagram Elements

GS4 Set of expected trace links between Sentences and Diagram Elements

The benchmark data set contains nine project versions in total. Each project includes files representing its NLSAD and its SAM. The data set was extended with informal diagrams sourced as closely as possible to the original sources. Six different diagram sources were added, covering seven of the nine project versions. The JabRef (JR) project versions are not used for the evaluation, because no diagrams exist. Each diagram file contains a single diagram consisting of an arbitrary amount of elements and is saved in a standard image format (e.g., PNG, JPEG). For the projects with SAD on a website like *Teammates* and *BigBlueButton*, the diagrams were extracted directly from the web page. JSON was chosen as a file format because it is readable for humans and can be parsed using Java libraries such as Jackson[1]. A JSON schema[2] was created to ensure that the files, as well as future extensions, provide a consistent structure to programs parsing the data set. Because the gold standard for the diagram-to-sentence trace links depends on the endpoints' gold standards, the diagram element and diagram-to-sentence trace link gold standard are combined in a single JSON structure. The structure of a gold standard JSON file is depicted in Figure 5.4 and explained below. A combination of attributes and relationship arrows shows the hierarchy and multiplicity due to space constraints.

GoldStandard Each file contains a *GoldStandard* object. The *diagrams* property of this object contains an array of *Diagram* objects. The array contains at least one *Diagram* as an element because the gold standard file should only be present if the project has corresponding diagrams.

Diagram A *Diagram* is identified by its *path* property, which contains the relative file path to its diagram image file as a string. The *boxes* property contains an array of *Box* objects. A *Diagram* needs to have at least one *Box*, because *Box* is considered to be the most generic diagram element, and empty diagrams are not considered. If this interpretation of diagrams changes in the future, this restraint may be relaxed to zero instead.

[1] See FasterXML/jackson
[2] See JSON-schema.org

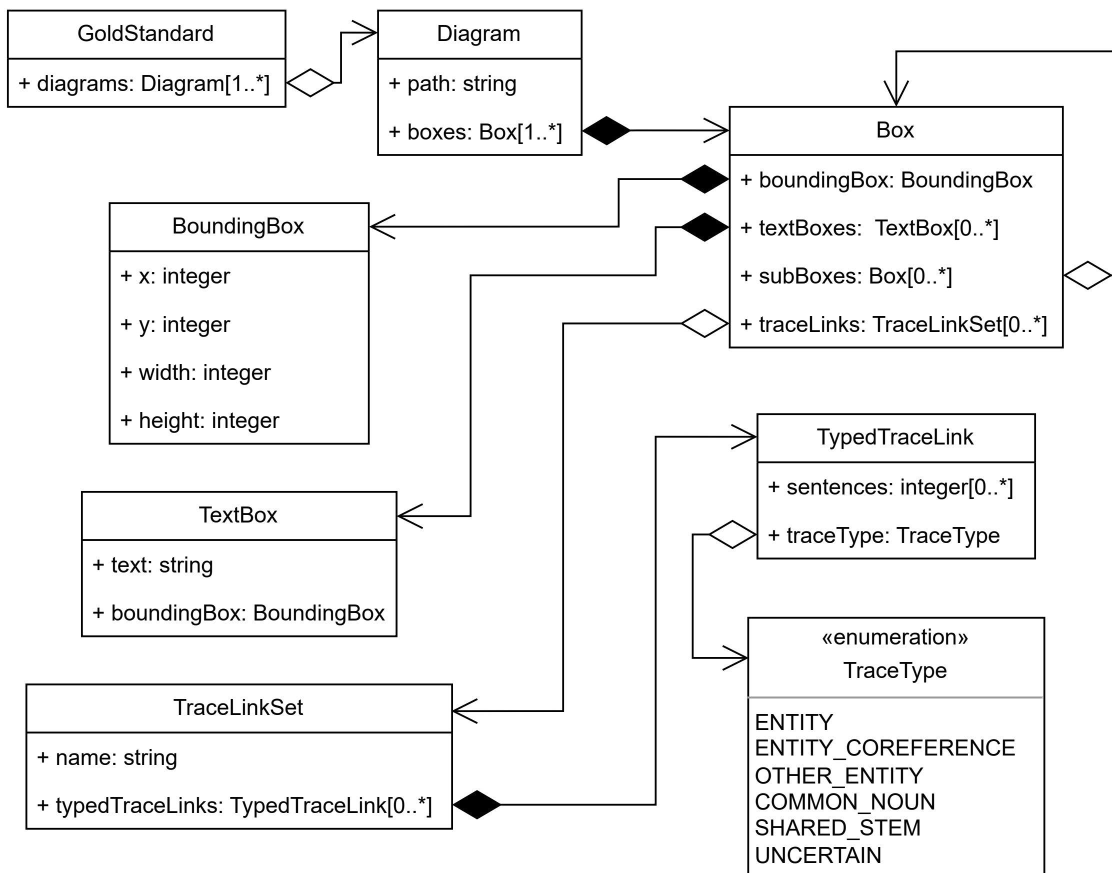


Figure 5.4: JSON schema in UML notation

Box A *Box* is the most generic form of diagram element contained by the gold standard. The *Box* is identified by its *BoundingBox* object, which semantically denotes the position and size of the element in the *Diagram* it is associated with. A *Box* may contain an arbitrary amount of *TextBox* objects and an arbitrary amount of *Box* sub-boxes that are contained within its *BoundingBox*. The *subBoxes* property was created to reflect the hierarchy that can be found in diagrams with nested elements. At last, the *Box* also contains an array of *TraceLinkSet* objects that describe the trace links that link to the *Box* as diagram element endpoint.

TextBox The *TextBox* object contains its text as a string, and the *BoundingBox* of its text. When compared with other *TextBox* objects from the same *Diagram*, it is uniquely identified by the *BoundingBox* of its text. Semantically, a *TextBox* contains only its diagram text and is part of the innermost *Box* surrounding it. A *TextBox* is descriptive and provides information about its parent diagram element. Therefore, a *TextBox* does not constitute a diagram element by itself.

TraceLinkSet A *TraceLinkSet* contains information about the trace links contained by its parent diagram element. The sentence numbers of the trace link refer to the gold standard text file with the given *name*. It is structured as the path to the file relative to

the gold standard project folder, e.g., "text_2018/teastore_2018_AB.txt" for a file from TeaStore-Historical. The *typedTraceLinks* property holds a set of *TypedTraceLink* objects.

TypedTraceLink The *TypedTraceLink* object contains trace links with a textual endpoint of a specific *TraceType*. The *sentences* property contains an array of integers. Each integer is the sentence number of a sentence from the file denoted in *name*. Sentence number indexing starts at one. As described previously, a diagram-to-sentence trace link is represented by the sentence number and the parent *Box*. The *traceType* property holds a string from the *TraceType* enum.

TraceType The *TraceType* enum is used to provide context to a trace link's textual endpoint. Table 5.1 provides an overview of the types. The *ENTITY* type semantically describes a trace link where the textual endpoint contains the same entity as the diagram element excluding coreferences. Coreferences are covered by the *ENTITY_COREFERENCE* type instead. All other types represent negatives and are not used directly in calculating the metrics but solely for debugging. If the sentence contains an entity with a similar name to the diagram element entity, the *OTHER_ENTITY* type is used. If we consider an ambiguously named entity "test", it becomes evident that entity names can be similar or, in some cases, identical to common nouns or other words that do not necessarily refer to the entity. Such ambiguities can lead to false positives in similarity-based TLR. Some of these negatives are added to the gold standard with the corresponding *COMMON_NOUN* and *SHARED_STEM* types to allow easier identification of the source of potential false positives. *UNCERTAIN* marks a potential trace link for discussion. *ENTITY* and *ENTITY_COREFERENCE* denote positives. *OTHER_ENTITY*, *COMMON_NOUN* and *SHARED_STEM* denote possible false positives. *UNCERTAIN* is used as a marker.

TraceType	**Description**
ENTITY	Trace link between a textual entity and a diagram element excluding coreferences
ENTITY_COREFERENCE	Trace link from a diagram element to a coreference of a textual entity
OTHER_ENTITY	False trace link between a diagram element and a different entity
COMMON_NOUN	False trace link between a diagram element and a non-entity noun
SHARED_STEM	False trace link between a diagram element and a non-entity word
UNCERTAIN	Mark a trace link for discussion

Table 5.1: Short description of the different trace types

The *GoldStandard*, *Diagram*, *BoundingBox*, *Box* and *TextBox* schemata cover the gold standard **GS3** as defined previously. **GS4** is covered by the extension with *TraceLinkSet*, *TypedTraceLink* and *TraceType*.

5.4 Diagram Text Traceability Link Recovery

After achieving goal G1, the question **Q1** concerning the Diagram Text TLR was evaluated. In the context of this section, trace link always refers to diagram-to-sentence trace links as described in Section 5.2 unless explicitly stated otherwise. The projects MediaStore (MS), TeaStore (TS), TEAMMATES (TM), BigBlueButton (BBB), as well as the corresponding historical projects, were used for the evaluation. JR was excluded from this and all other parts of the evaluation because there were no diagram sources for the project. For this part of the evaluation, no prior work exists. Therefore, the evaluation focuses solely on the metrics and is not comparative. All metrics introduced in Section 5.1 were used for this part of the evaluation. The metrics rely on the concept of TP, TN, FP, and FN, which need to be defined for this context before they can be applied. Two trace links are considered equal if they have matching endpoints. TP can thus be defined as the number of trace links contained in the gold standard that match with trace links generated by the approach. Subsequently, FP is the number of trace links generated by the approach that are not contained in the gold standard. FN is calculated as the number of trace links from the gold standard **GS4** not included in the set of trace links generated by the approach. The calculation of TN is based on the previous variables and the number of possible endpoint combinations a trace link can have. The values can be calculated using the equations 5.2 with the following variable assignment:

$$PP = TL_A \quad \text{Set of trace links reported by the approach}$$
$$AP = TL_{GS4} \quad \text{Set of trace links in gold standard } \mathbf{GS4}$$
$$Total = \#Sentences * DE_{GS3}$$

where:

$\#Sentences$ is the amount of sentences in the SAD

DE_{GS3} is the set of diagram elements in gold standard **GS3**

The average and weighted average were calculated for all metrics as in previous work [17]. The weight is directly proportional to each project's number of expected trace links, i.e., TL_{GS2}. Table 5.2 shows the evaluation result for each project with the historical version being annotated "-H". The results for Precision (P) can be classified as excellent on both weighted and unweighted averages using the classification of Hayes et al. [14]. The only noticeable outlier is TM and TEAMMATES-Historical (TM-H) with a high amount of FP. A qualitative assessment of the FP trace links shows that the approach struggles with textual entities named after commonly used words. For example, one of the diagram elements of TM is called "test". It also struggles with determining the proper entity if multiple entities share the same prefix. This is especially the case for sub-packages in package diagrams due to naming conventions.

According to the classification scheme of Hayes, the Recall (R) is excellent on average and good when considering the weighted average. TM and TM (Historical) again feature the worst results. The trace links are created assuming multiple diagram elements cannot refer to the same entity in the text unless the diagram elements are identical. This can cause a FP to overshadow a positive, leading to a FN, which reduces the Recall.

Project	P	R	F_1	Acc	Spec	Φ	Φ_N
MS	0.87	0.93	0.90	0.99	0.99	0.89	0.92
TS	1.00	0.74	0.85	0.97	1.00	0.85	1.00
TM	0.60	0.67	0.63	0.98	0.99	0.62	0.66
BBB	0.79	0.72	0.75	0.97	0.99	0.74	0.77
TS-H	1.00	0.92	0.96	0.99	1.00	0.95	1.00
TM-H	0.63	0.71	0.66	0.98	0.99	0.66	0.70
BBB-H	0.73	0.91	0.81	0.98	0.98	0.80	0.90
Average	0.80	0.80	0.79	0.98	0.99	0.79	0.85
w. Average	0.72	0.76	0.74	0.98	0.99	0.73	0.78

Table 5.2: Results for Diagram Text TLR with manually extracted Diagram Elements

The F_1-score is excellent for both the weighted and unweighted average in reference to the lowest Precision and Recall scores classified as excellent by Hayes et al. Both Accuracy (Acc) and Specificity (Spec) are within 3pp of 1.00 for all projects. The Φ_N coefficient suggests a strong positive correlation for both the average and weighted average, according to Dancey and Reidy [1]. In summary, the performance ranges from good to excellent but suffers from ambiguously named diagram elements. This answers evaluation question **Q1**.

Project	P	R	F_1	Acc	Spec	Φ	Φ_N
MS	0.81	0.93	0.87	0.98	0.98	0.86	0.92
TS	0.65	0.74	0.69	0.93	0.95	0.65	0.71
TM	0.37	0.25	0.30	0.97	0.99	0.29	0.36
BBB	0.76	0.61	0.67	0.97	0.99	0.66	0.74
TS-H	0.77	0.92	0.84	0.96	0.96	0.82	0.91
TM-H	0.51	0.39	0.44	0.98	0.99	0.43	0.50
BBB-H	0.69	0.77	0.73	0.97	0.98	0.71	0.75
Average	0.65	0.66	0.65	0.97	0.98	0.63	0.70
w. Average	0.57	0.52	0.54	0.97	0.98	0.53	0.59

Table 5.3: Results for Diagram Text TLR with automatic Diagram Recognition

After goal G3 was achieved, the Diagram Text TLR was evaluated using the automatically extracted diagram elements from the *Diagram Recognition* stage to answer **Q2**. The process for calculating the metrics was identical to the previous process. The weighted and unweighted average Precision (P) decreased by 15pp. According to the classification scheme, the achieved Precision is still considered excellent. The reduced Precision is caused by an increased amount of FPs, as well as an increased amount of FNs. Based on observation, the recognition misreads labels, including cases of truncated words, spelling errors, or missed labels. These errors can only be partially resolved by the similarity-based approach. A single truncated word can lead to many FPs. For example, consider the TS

component “Image-Provider”. The component name contains a line break in the diagram due to space restrictions. If the second part of the word is missed, a component “Image” is identified, which will lead to a FP every time the common word “Image” is mentioned in the text.

The average Recall (R) decreased by 14pp, and the weighted average Recall decreased by 24pp. The Recall decreased by 42pp for TM and by 32pp for TM-H in particular. A misread label can push the similarity low enough to prevent a trace link, which subsequently lowers Recall as observed. Both versions feature colored diagrams with nested diagram elements, complex shapes, and low contrast. These variables seem to negatively impact the OCR and OD. This causes some diagram elements and labels to be missed. When comparing the Recall to the classification scheme, one result is acceptable, two are good, and two are excellent. The TM results are below acceptable. Overall, the results for Recall are acceptable on average and below acceptable on weighted average.

The F_1-score is excellent on average and good for the weighted average. Φ_N suggests moderate correlation according to the classification scheme. In summary, the combined approach can resolve reasonable spelling mistakes, but further work is required to improve the OCR, OD, and robustness. However, it was expected that the approach would perform worse than ArDoCo’s SAD SAM TLR when viewed in isolation.

5.5 Impact on ArDoCo

In this part of the evaluation, ArDoCo’s End-to-End performance is evaluated to answer **Q3**. Similar to the evaluations in prior work, a twofold evaluation is performed to determine ArDoCo’s SAD to SAM TLR capabilities, as well as its Inconsistency Detection capabilities. The SAD to SAM TLR is evaluated in Subsection 5.5.1. The impact on ArDoCo’s Missing Model Element (MME) detection capabilities is evaluated in Subsection 5.5.2. Subsection 5.5.3 describes the Unmentioned Model Element (UME) detection evaluation. All evaluated versions are based on ArDoCo release 0.22 to ensure comparability. The evaluation is performed using the existing evaluation implementation with modifications to allow running a different pipeline configuration. The baseline is denoted *ArDoCo* in the tables. The tables include *ArDoCo (Altered)* as an altered version of ArDoCo, which does not use diagrams but includes all changes made to other stages or globally. This configuration is evaluated to determine how much of the impact can be traced back to the diagrams and to ensure that the changes do not negatively impact ArDoCo for projects without diagrams. The approach is named ERID in the tables and combines the changes from *ArDoCo (Altered)* with the use of diagrams as described in Chapter 3. Additionally, the evaluation distinguishes between *ERID (Mock)* and *ERID*. The distinction is made because *ERID (Mock)* eliminates all inaccuracies introduced by the diagram recognition by using the gold standard *DiagramElements*. Therefore, *ERID (Mock)* is more akin to a best-case scenario and shows the capabilities of the approach independent from the diagram recognition implementation. A **Bold** result is best for a specific project and metric. Generally, greater can be considered better for all metrics used. To prevent bias the comparison of metrics is between $min(\textit{ERID (Mock)}, \textit{ERID})$ and *ArDoCo (Altered)* unless otherwise indicated.

5.5.1 SAD to SAM Traceability Link Recovery

This subsection focuses on comparing the approach's impact on ArDoCo's Software Architecture Documentation (SAD) to Software Architecture Model (SAM) Traceability Link Recovery (TLR) capabilities. The results are depicted in Table 5.4. TP, FP, TN, and FN are defined by comparing a reported trace link to the trace links contained in the gold standard **GS1**. The endpoints of a SAD to SAM trace link are a sentence number and a model element. A reported trace link is a true positive if a trace link with the same endpoints is contained in the gold standard. It is considered a false positive if no corresponding trace link with the same endpoints exists in the gold standard. If the approach does not report a trace link from the gold standard, it is considered a false negative. Every possible trace link, which is neither reported nor contained in the gold standard, is a true negative. TN is calculated by calculating the amount of possible endpoint combinations and subtracting TP, FP, and FN. The values can be calculated using the equations 5.2 with the following variable assignment:

$$PP = TL_A \qquad \text{Set of trace links reported by the approach}$$
$$AP = TL_{GS1} \qquad \text{Set of trace links in gold standard } \mathbf{GS1}$$
$$Total = \#Sentences * \#ModelElements$$

where:

$\#Sentences$ is the amount of sentences in the SAD

$\#ModelElements$ is the amount of model elements in the SAM

ArDoCo's SAD to SAM trace links consist of a model element unique identifier (UID) and a sentence number in the evaluation. For example, consider the text with ten sentences $S = \{1, .., 10\}$ and the model elements A and B where the trace links are formatted as tuples. The gold standard contains the model element UID sentence number tuples $AP = \{(A, 1), (B, 7)\}$. The approach reports the tuples $PP = \{(A, 1), (B, 3)\}$. The amount of classified inputs is $Total = |S| * |\{A, B\}| = 20$. TP is the number of tuples contained by both AP and PP, which is only $|(A, 1)| = 1$. The trace link from B to sentence 3 is not part of the actual positives, so $|(B, 3)| = 1$ is the number of false positives. The trace link $(B, 7)$ is part of the gold standard but not reported and constitutes a false negative. Therefore, $FN = 1$. TN can then be calculated as $TN = 20 - (1 + 1 + 1) = 17$.

Each project is weighted by the amount of Actual Positives from the gold standard to calculate the weighted average. This is to prevent projects with few trace links from disproportionately affecting the overall results. An average where every project is weighted equally is provided for comparison. According to prior work, ArDoCo outperforms similar TLR approaches and performs excellently [17]. ArDoCo's TLR as a baseline has excellent Precision (P), Recall (R), and F_1-score. Therefore, the approach was not expected to noticeably improve the overall TLR results. This is consistent with the results from Table 5.4. However, changes are visible for some individual projects.

For MS Precision worsened by 23pp from the *ArDoCo* baseline. Due to the implementation of abbreviation disambiguation, ArDoCo is now capable of resolving some of the abbreviations in the MS project, such as “DB” meaning “Database”. However, this disam-

biguation has adverse side effects. “Database” is used as both a common word and the name of a component in the MS text. ArDoCo cannot differentiate between the use as a common word and a component. Because of this, a trace link is created between the “DB” model element and every mention of “Database”. The additional TPs increase the Recall of MS by 17pp (+24%). However, this also leads to the creation of FPs for sentences using the common word. The F_1-score of MS, the harmonic mean of both metrics, is not negatively impacted and improves by 1pp. This is because Recall was relatively low compared to Precision, so the overall change to both metrics is considered an improvement by the metric. The results for *ERID (Mock)* and *ERID* are identical to *ArDoCo (Altered)* for MS. This is to be expected because there is no mechanism by which the approach impacts ArDoCo’s capability to differentiate between the common use of a word and the use as a component. For the other projects, the changes between *ArDoCo* and *ArDoCo (Altered)* are negligible.

In the case of TM and TM-H, a large amount of FPs negatively impacts the Precision of the baseline. Many of these FPs are because of computer- or software-related words, which are falsely identified as textual entities. The approach disaffirms many of these textual entities because they are not contained in the diagrams. This leads to the observed reduction in FPs. For TM Precision improves by 18pp (+32%), which subsequently improves the F_1-score by 12pp (+17%). TM-H’s Precision increased by 13pp (+22%) with a F_1improvement of 7pp (+10%).

TS and TeaStore-Historical (TS-H) had a perfect precision score for all configurations. This leaves MS, BBB, and BigBlueButton-Historical (BBB-H) for possible improvements to the precision. All three projects have a precision above 0.77 for *ArDoCo (Altered)*. As previously explained, the approach cannot mitigate the worsening of Precision for MS. Similarly to MS, BBB’s false positives stem from words, which have different meanings depending on the context in which they are used. The BBB-H project is an outlier in general because of the large version gap between the BBB-H model and the text. The text is from 2015, but the Palladio Component Model model is from 2021. Conceptually, the approach affirms *RecommendedInstances* that are part of the diagrams and disaffirms those that are not. Therefore, the approach is not beneficial for diagrams that are inconsistent with the model and may even be counterproductive. BBB-H is the only project where Precision worsened. Precision decreased by 3pp (-4%), which negatively impacts the F_1-score by 1pp (-2%).

The approach does not impact Recall (R) for any of the projects. Accuracy (Acc) and Specificity (Spec) remain within 2pp of 1.00. The impact on Φ_N slightly improves for the weighted average and is still interpreted as a strong correlation [1]. In summary, the approach performs as expected. The approach is capable of improving the Precision of projects that have low Precision due to falsely identified *RecommendedInstances* with negligible impact on all other projects. All metrics have improved or remained equal for the weighted average.

Project	Configuration	P	R	F_1	Acc	Spec	Φ	Φ_N
MS	*ArDoCo*	**1.00**	0.62	0.77	**0.98**	**1.00**	**0.78**	**1.00**
	ArDoCo (Altered)	0.77	**0.79**	**0.78**	0.97	0.99	0.77	0.78
	ERID (Mock)	0.77	**0.79**	**0.78**	0.97	0.99	0.77	0.78
	ERID	0.77	**0.79**	**0.78**	0.97	0.99	0.77	0.78
TS	*ArDoCo*	**1.00**	**0.74**	**0.85**	**0.99**	**1.00**	**0.85**	**1.00**
	ArDoCo (Altered)	**1.00**	**0.74**	**0.85**	**0.99**	**1.00**	**0.85**	**1.00**
	ERID (Mock)	**1.00**	**0.74**	**0.85**	**0.99**	**1.00**	**0.85**	**1.00**
	ERID	**1.00**	**0.74**	**0.85**	**0.99**	**1.00**	**0.85**	**1.00**
TM	*ArDoCo*	0.56	**0.88**	0.68	0.97	0.98	0.69	**0.88**
	ArDoCo (Altered)	0.56	**0.88**	0.68	0.97	0.98	0.69	**0.88**
	ERID (Mock)	**0.74**	**0.88**	**0.80**	**0.99**	**0.99**	**0.80**	**0.88**
	ERID	**0.74**	**0.88**	**0.80**	**0.99**	**0.99**	**0.80**	**0.88**
BBB	*ArDoCo*	**0.88**	**0.83**	**0.85**	**0.99**	**0.99**	**0.84**	**0.87**
	ArDoCo (Altered)	**0.88**	**0.83**	**0.85**	**0.99**	**0.99**	**0.84**	**0.87**
	ERID (Mock)	**0.88**	**0.83**	**0.85**	**0.99**	**0.99**	**0.84**	**0.87**
	ERID	**0.88**	**0.83**	**0.85**	**0.99**	**0.99**	**0.84**	**0.87**
TS-H	*ArDoCo*	**1.00**	**0.93**	**0.97**	**1.00**	**1.00**	**0.96**	**1.00**
	ArDoCo (Altered)	**1.00**	**0.93**	**0.97**	**1.00**	**1.00**	**0.96**	**1.00**
	ERID (Mock)	**1.00**	**0.93**	**0.97**	**1.00**	**1.00**	**0.96**	**1.00**
	ERID	**1.00**	**0.93**	**0.97**	**1.00**	**1.00**	**0.96**	**1.00**
TM-H	*ArDoCo*	0.52	0.70	0.60	0.97	0.98	0.59	0.68
	ArDoCo (Altered)	0.57	**0.76**	0.65	**0.98**	0.98	0.65	**0.75**
	ERID (Mock)	**0.73**	**0.76**	**0.74**	**0.98**	**0.99**	**0.74**	**0.75**
	ERID	0.69	**0.76**	0.72	**0.98**	**0.99**	0.71	**0.75**
BBB-H	*ArDoCo*	**0.81**	**0.62**	**0.70**	**0.98**	**0.99**	**0.70**	**0.80**
	ArDoCo (Altered)	**0.81**	**0.62**	**0.70**	**0.98**	**0.99**	**0.70**	**0.80**
	ERID (Mock)	0.78	**0.62**	0.69	**0.98**	**0.99**	0.68	0.77
	ERID	0.78	**0.62**	0.69	**0.98**	**0.99**	0.68	0.77
Average	*ArDoCo*	0.82	0.76	0.77	**0.98**	**0.99**	0.77	**0.89**
	ArDoCo (Altered)	0.80	**0.79**	0.78	**0.98**	**0.99**	0.78	0.87
	ERID (Mock)	**0.84**	**0.79**	**0.81**	**0.98**	**0.99**	**0.81**	0.86
	ERID	**0.84**	**0.79**	**0.81**	**0.98**	**0.99**	0.80	0.86
weighted Average	*ArDoCo*	0.81	0.79	0.78	0.98	**0.99**	0.76	0.78
	ArDoCo (Altered)	0.79	**0.81**	0.79	0.98	**0.99**	0.77	0.80
	ERID (Mock)	**0.84**	**0.81**	**0.82**	**0.99**	**0.99**	**0.82**	**0.83**
	ERID	**0.84**	**0.81**	**0.82**	**0.99**	**0.99**	0.81	0.82

Table 5.4: Results for Architecture Document to Architecture Model TLR

5.5.2 Missing Model Element Inconsistency Detection

This part of the evaluation focuses on detecting textual entities that should be contained in the Software Architecture Model (SAM) but are not. Such entities are classified as Missing Model Element (MME). The evaluation of ArDoCo's Missing Model Element (MME) Inconsistency Detection capabilities was performed using the same benchmark data set as previous work. The results in Table 5.5 use the same metric and format as the evaluation in the last subsection. The approach to find MMEs is the same approach used in previous work [17]: Each project is run $|\{ModelElements(Project)\}|$ times. A single model element is removed before every run. The existing trace links to the removed model element subsequently indicate MME inconsistencies in the run. This allows using all projects from the data set, despite none of the projects containing any MMEs by default, and allows reusing ArDoCo's TLR gold standard to determine MMEs. In this subsection, True Positives (TPs) are MME inconsistencies, which cover a sentence that mentions a removed model element. False Positives (FPs) are inconsistencies that cover sentences that do not mention the removed element. False Negatives (FNs) are indicated by a sentence mentioning a model element but not being covered by a MME inconsistency. Lastly, each sentence that does not mention the removed element and is not contained by any MME constitutes a TN. The values can be calculated for a run using the equations 5.2:

$$
\begin{aligned}
ME &= \text{model element} \\
RM &= \text{removed model element} \\
PP &= |\bigcup_{i=1}^{n} S(PMME_i)| \\
AP &= |S_{GS1}(RM)| \\
Total &= \#Sentences
\end{aligned}
$$

where:

$\#Sentences$ is the amount of sentences in the SAD

$PMME_i$ is the i-th reported MME (of n) in the run

RM is the removed model element

$S(PMME_i)$ are the sentences containing the reported MME

$S_{GS1}(ME)$ sentences in gold standard **GS1** linking to model element

As an example, consider a text with ten sentences $S = \{1, .., 10\}$ and a single model element A with $S_{GS1}(A) = \{1, 2\}$. In the first run $RM = A$ is removed. Two MMEs are reported with $S(PMME_1) = \{1, 2\}$ and $S(PMME_2) = \{7\}$. Then $PP = \{1, 2, 7\}$ and $AP = \{1, 2\}$.

Since a singular result is required for each project, a weighted average is calculated over the runs. The weight of each run is directly proportional to the number of sentences in the text where the removed model element is mentioned normalized by the sum of all weights. Additionally, an overall average and a weighted average are calculated over all runs from all projects. The weights for the overall weighted average are calculated as previously described by using the total weight of all runs combined for normalization.

Compared to ArDoCo as a baseline, the approach achieves equal or improved precision for six out of seven projects, with BBB-H being an outlier. However, BBB-H is an outlier in general, with the lowest scores across all metrics for both the approach and the baseline. The reasons for this are explained in Subsection 5.5.1. However, the overall metric changes for BBB-H are negligible relative to the already poor performance of the baseline.

TM and TM-H are the largest projects contained in the data set, and while the baseline achieves high Recall (R), both projects suffer from low Precision (P). The approach greatly increases the Precision for both projects. The MME detection is based on the SAD to SAM TLR, which improved for both projects. It is also affected by the MDE detection because a MDE lowers the confidence of a *RecommendedInstance*. This can lead to the *RecommendedInstance* being filtered out. The Precision increases by 44pp from 18% to 62% for TM independent of whether the mock was used. For TM-H, an increase of 14pp is achieved using the mock. A 29pp increase is observed when using *ERID*. The discrepancy is due to mistakes in the automatic diagram extraction, which coincidentally disaffirms some *RecommendedInstances* that were responsible for False Positives (FPs). This result demonstrates the importance of evaluating with and without the mock.

Regarding Precision, the change between the baseline and *ArDoCo (Altered)* is negligible for all projects except MS. The altered version achieves more than twice as high Precision with an increase of 31pp. As previously explained in Subsection 5.5.1, MS contains abbreviations in its model, which the baseline can not resolve. For example, the *RecommendedInstance* for "Database" is falsely detected as a MME in the baseline, despite the existence of the corresponding "DB" model element. This is not the case in the altered version. The approach increases the Precision further by 40pp, leading to a total improvement of 69pp (or about 328%).

Overall, Precision increased by 28pp (+66%) on average and by 31pp (+79%) on weighted average. Precision can be increased by increasing the amount of TPs or reducing the amount of FPs. Trying to reduce FPs may lead to the reduction of TPs, which negatively impacts the Recall. However, the approach reduces Recall by only -3pp (-6%) on average and -3pp (-5%) on weighted average, which is a negligible change when compared to the overall Precision improvement. The approach achieves the highest Precision increase for TS-H with 84pp (+525%) while retaining its excellent recall with a decrease of -6pp (-6%). This result and the overall result show that the approach can reduce the amount of FPs without affecting the TPs disproportionately.

The overall improvement of the F_1-score further corroborates the previous statement. The F_1-score increases by 12pp (+36%) on average and 20pp (+55%) on weighted average. BBB is the only project with a reduced F_1-score of -3pp (-10%). BBB has a relatively high Precision and low Recall compared to the other projects. Therefore, the negative effect on Recall affects BBB relatively strongly with a relatively weak positive effect on Precision, which in sum has a negative effect on the F_1-score.

Accuracy (Acc) increased or remained equal for all projects except BBB-H. In the case of BBB-H, the approach identifies slightly more TPs but suffers from more FPs ($\leftrightarrow$ less TNs). This negatively impacts the accuracy by -7pp (-9%). The Specificity (Spec) of BBB-H decreased by -8pp (-9%) for the same reason. However, the approach achieves an overall increase of 9pp (+10%) on average and 15pp (+18%) on weighted average, pushing the Accuracy over 90% in both cases. The Specificity, or *True Negative Rate*, increases by 11pp

(+12%) on average and by 17pp (+21%) on weighted average. In both cases, the overall Specificity is 96%.

The Φ_N-coefficient decreases by 0.02 on average but improves by 0.03 on weighted average. However, the changes are too small to affect the interpretation of Φ_N [1].

In summary, the approach positively affects Precision, F_1-score, Accuracy, and Specificity with negligible changes to Recall and Φ_N.

Project	Configuration	P	R	F_1	Acc	Spec	Φ	Φ_N
MS	*ArDoCo*	0.21	**0.79**	0.33	0.70	0.69	0.23	0.68
	ArDoCo (Altered)	0.50	0.72	0.57	0.91	0.92	0.51	0.69
	ERID (Mock)	**0.90**	0.72	**0.68**	**0.95**	**0.96**	**0.72**	**0.74**
	ERID	**0.90**	0.72	**0.68**	**0.95**	**0.96**	**0.72**	**0.74**
TS	*ArDoCo*	0.96	**0.70**	0.79	**0.96**	**1.00**	0.81	0.95
	ArDoCo (Altered)	0.96	**0.70**	0.79	**0.96**	**1.00**	0.81	0.95
	ERID (Mock)	**1.00**	**0.70**	**0.80**	**0.96**	**1.00**	**0.83**	**1.00**
	ERID	0.96	**0.70**	0.79	**0.96**	**1.00**	0.81	0.95
TM	*ArDoCo*	0.18	**0.75**	0.28	0.85	0.85	0.29	**0.70**
	ArDoCo (Altered)	0.18	**0.75**	0.28	0.85	0.85	0.29	**0.70**
	ERID (Mock)	**0.62**	0.71	**0.54**	**0.97**	**0.97**	**0.58**	0.69
	ERID	**0.62**	0.71	**0.54**	**0.97**	**0.97**	**0.58**	0.69
BBB	*ArDoCo*	0.89	**0.46**	**0.43**	**0.96**	**0.99**	0.54	0.65
	ArDoCo (Altered)	0.89	**0.46**	**0.43**	**0.96**	**0.99**	0.54	0.65
	ERID (Mock)	**0.94**	0.38	0.39	**0.96**	**0.99**	**0.55**	**0.82**
	ERID	**0.94**	0.38	0.39	**0.96**	**0.99**	**0.55**	**0.82**
TS-H	*ArDoCo*	0.16	**0.98**	0.28	0.38	0.29	0.15	0.94
	ArDoCo (Altered)	0.16	**0.98**	0.28	0.37	0.28	0.14	0.94
	ERID (Mock)	**1.00**	0.92	**0.92**	**1.00**	**1.00**	**0.95**	**1.00**
	ERID	0.84	0.92	0.88	0.98	0.99	0.87	0.91
TM-H	*ArDoCo*	0.17	0.63	0.26	0.86	0.87	0.26	0.57
	ArDoCo (Altered)	0.19	**0.70**	0.29	0.87	0.88	0.30	0.65
	ERID (Mock)	0.31	**0.70**	0.42	0.94	0.94	0.44	0.67
	ERID	**0.46**	**0.70**	**0.50**	**0.96**	**0.96**	**0.54**	**0.68**
BBB-H	*ArDoCo*	**0.09**	0.18	**0.11**	**0.81**	**0.87**	**0.02**	**0.04**
	ArDoCo (Altered)	**0.09**	0.18	**0.11**	**0.81**	**0.87**	**0.02**	**0.04**
	ERID (Mock)	0.07	**0.21**	**0.11**	0.74	0.79	-0.00	-0.01
	ERID	0.07	**0.21**	**0.11**	0.74	0.79	-0.00	-0.01
Average	*ArDoCo*	0.37	**0.56**	0.28	0.80	0.81	0.27	0.53
	ArDoCo (Altered)	0.42	0.55	0.33	0.84	0.85	0.31	**0.55**
	ERID (Mock)	**0.70**	0.52	**0.45**	**0.93**	**0.96**	**0.44**	0.53
	ERID	0.68	0.52	**0.45**	**0.93**	**0.96**	**0.44**	0.54
weighted Average	*ArDoCo*	0.36	0.66	0.33	0.77	0.76	0.23	0.57
	ArDoCo (Altered)	0.39	**0.67**	0.36	0.79	0.79	0.24	0.58
	ERID (Mock)	**0.70**	0.64	**0.56**	**0.94**	**0.96**	0.47	**0.61**
	ERID	0.69	0.64	**0.56**	**0.94**	**0.96**	**0.48**	**0.61**

Table 5.5: Results for ArDoCo's Missing Model Element (MME) Inconsistency Detection

5.5.3 Unmentioned Model Element Inconsistency Detection

A model element is classified as Unmentioned Model Element (UME) if no corresponding textual entity exists. This subsection focuses on the evaluation of ArDoCo's UME detection capabilities using the different configurations. The results are depicted in Table 5.6. UME detection is performed by running ArDoCo's Inconsistency Detection once per project and comparing the reported UMEs to the UME gold standard **GS2**. UMEs are uniquely identified by their model element UID. A UME is classified True Positive (TP) if its model element is contained in the set of model elements from the gold standard. If this is not the case, it is considered False Positive (FP). If a model element is part of the gold standard, but no UME with a matching model element is reported, the absence of a UME for the model element is classified as False Negative (FN). The number of True Negatives (TNs) is the number of model elements without a reported UME, which are neither reported nor contained in the gold standard and is calculated using the equations 5.2 with variable assignment:

$$PP = UME_A \quad \text{Set of reported Unmentioned Model Elements}$$
$$AP = UME_{GS1} \quad \text{Set of Unmentioned Model Elements from } \textbf{GS2}$$
$$Total = \#ModelElements \quad \#ModelElements \text{ is the amount of model elements}$$

The metrics in Table 5.6 show that the values did not change for all projects except MediaStore (MS). The approach is designed to adjust the confidence in textual entities depending on whether they are present in the diagrams. Reducing the confidence in a *RecommendedInstance* can only impact the UME if it is a falsely identified textual entity that coincidentally covers an unrelated model element. This scenario does not occur in the data set. By increasing the confidence in a *RecommendedInstance*, a textual entity that was falsely filtered due to low confidence may be preserved, which could prevent a UME FP in theory. However, this scenario does not occur in the data set either. In general, ArDoCo performs excellently at detecting UMEs, with five projects achieving a perfect score on all metrics. The two projects that do not have perfect scores are MS and BBB-H.

MS has perfect Recall (R), but falsely reports "DB" and "FileStorage" as UME, when using the *ArDoCo* baseline. The approach can not impact the "FileStorage" UME because the naming is inconsistent between the model and text. "FileStorage" is referred to as "DataStorage" in the text and diagrams. The approach affirms the "DataStorage" *RecommendedInstance* but can not resolve the inconsistent naming. The falsely reported "DB" UME is no longer reported by *ArDoCo (Altered)*, *ERID (Mock)*, and *ERID* because the abbreviation disambiguation is capable of identifying the corresponding "Database" *RecommendedInstance* in the text. This improves the Precision by 13pp (+19%) and the F_1-score by 9pp (+11%) without affecting other metrics.

Due to its large version gap, the BBB-H project has the worst metrics of all projects for UME detection. As explained in Subsection 5.5.1, the approach cannot resolve this issue. In summary, the approach performs as expected and does not negatively impact the UME detection capabilities of ArDoCo. Furthermore, the abbreviation disambiguation that was implemented positively affects the UME detection for MediaStore.

Project	Configuration	P	R	F_1	Acc	Spec	Φ	Φ_N
MS	*ArDoCo*	0.67	**1.00**	0.80	0.95	0.94	0.79	**1.00**
	ArDoCo (Altered)	**0.80**	**1.00**	**0.89**	**0.97**	**0.97**	**0.88**	**1.00**
	ERID (Mock)	**0.80**	**1.00**	**0.89**	**0.97**	**0.97**	**0.88**	**1.00**
	ERID	**0.80**	**1.00**	**0.89**	**0.97**	**0.97**	**0.88**	**1.00**
TS	*ArDoCo*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ArDoCo (Altered)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID (Mock)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
TM	*ArDoCo*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.00**	**1.00**
	ArDoCo (Altered)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.00**	**1.00**
	ERID (Mock)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.00**	**1.00**
	ERID	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.00**	**1.00**
BBB	*ArDoCo*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ArDoCo (Altered)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID (Mock)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
TS-H	*ArDoCo*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ArDoCo (Altered)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID (Mock)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
TM-H	*ArDoCo*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ArDoCo (Altered)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID (Mock)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
	ERID	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
BBB-H	*ArDoCo*	**0.50**	**0.75**	**0.60**	**0.94**	**0.96**	**0.58**	**0.73**
	ArDoCo (Altered)	**0.50**	**0.75**	**0.60**	**0.94**	**0.96**	**0.58**	**0.73**
	ERID (Mock)	**0.50**	**0.75**	**0.60**	**0.94**	**0.96**	**0.58**	**0.73**
	ERID	**0.50**	**0.75**	**0.60**	**0.94**	**0.96**	**0.58**	**0.73**
Average	*ArDoCo*	0.86	**0.96**	0.88	**0.98**	0.98	0.88	**0.95**
	ArDoCo (Altered)	**0.88**	**0.96**	**0.90**	**0.98**	**0.99**	**0.90**	**0.95**
	ERID (Mock)	**0.88**	**0.96**	**0.90**	**0.98**	**0.99**	**0.90**	**0.95**
	ERID	**0.88**	**0.96**	**0.90**	**0.98**	**0.99**	**0.90**	**0.95**
weighted Average	*ArDoCo*	0.84	**0.95**	0.86	**0.98**	0.98	0.84	**0.90**
	ArDoCo (Altered)	**0.87**	**0.95**	**0.88**	**0.98**	**0.99**	**0.86**	**0.90**
	ERID (Mock)	**0.87**	**0.95**	**0.88**	**0.98**	**0.99**	**0.86**	**0.90**
	ERID	**0.87**	**0.95**	**0.88**	**0.98**	**0.99**	**0.86**	**0.90**

Table 5.6: Results for ArDoCo's Unmentioned Model Element Inconsistency Detection

5.6 Threats to Validity

To assess threats to validity, the guidelines for conducting and reporting case study research in software engineering by Runeson and Höst [36] are used in this section. The four separate evaluation parts imply multiple threats to validity.

Construct Validity Construct validity concerns whether the measures represent what the researcher intended to measure. The evaluation is backed by quantitative measures commonly used in TLR [17]. The evaluation is performed using the same process as previous work. By comparing to the gold standards, the results are quantified without requiring a subjective interpretation. This reduces bias and the possibility of misinterpreting a construct. To ensure that measures are not distorted by changes to the general functionality of ArDoCo outside of the approach, these changes are separated into the different configurations used by the evaluation.

Internal Validity Common threats to the *Internal Validity* are *History*, *Maturation* and *Instrumentation* [43]. The threats are related to ArDoCo, the various gold standards, and the approach. ArDoCo is an evolving software that received multiple updates during this work. This affects the program itself (*ArDoCo Core*), as well as the data set and gold standards. A *History* effect occurs if ArDoCo's capabilities changed outside of the approach between the separate versions used to compare the capabilities. To prevent this, ArDoCo 0.22 is used as a shared baseline version in the evaluation. Without countermeasures *Maturation* of ArDoCo's Benchmark data set and the contained gold standards may also affect the evaluation outcome. This can occur because the gold standard is updated to add missing information or remove wrong information. During the duration of this work, the gold standard was integrated into the *ArDoCo Core* project, expanded with code models, and its file structure was restructured. However, because it is now integrated into *ArDoCo Core* and all evaluations share the same baseline, they all share the same Benchmark data set baseline.

The *Instrumentation* effect can occur if the way measurements of metrics are performed changes between evaluations. A shared measurement process is used to mitigate this effect where possible. This is achieved by using the same evaluation test suite across all versions in the case of the End-to-End evaluation in Section 5.5. The evaluation test suite is adapted to use the custom pipelines and extended projects for the different configurations evaluated by this work. However, the adaption does not change how the metrics are calculated. The evaluation of the Diagram Text Traceability Link Recovery is not comparative because prior work could not perform Diagram Text TLR. Still, the same formulas are used to calculate the metrics.

The creation of the **GS3** and **GS4** by the author is a threat to the *Internal Validity* of the evaluation because it may introduce a bias. To prevent this bias, the gold standard creation described in Section 5.3 was performed at the beginning of the implementation. If uncertainty about the validity of an entry to the gold standard arose, it was discussed. The selection of diagrams was determined with the primary advisor, and all diagrams are sourced from the original data set sources. The diagram element gold standard **GS3** contains all diagram elements and their associated text boxes, irrespective of whether or not they are relevant to the approach. This includes information detrimental to the approach, such as text boxes containing software terminology. If a corresponding model

element to a diagram element exists, the model element trace links are used as a baseline to mitigate the bias for the Diagram Text TLR.

External Validity Whether the results achieved by the approach are valid under different circumstances is a threat to validity. This is referred to as *External Validity* and transferability [9]. The informal diagrams cover shapes like hexagons, boxes, and packages, ranging from monochrome to colored. However, the implementation of the approach does not rely on color or a particular geometric shape. Therefore, the approach should be able to work for any diagram element containing text. However, not every shape can be detected due to the limitations of the diagram recognition. This could be improved, but the OD algorithm is not the subject of this work. The overall transferability also depends on the transferability of ArDoCo. The projects from the Benchmark data set differ in architecture style, age, and size to reduce bias [17]. This promotes transferability by mitigating overfitting the approach to a particular project's circumstances.

Reliability The *Reliability* is the extent to which different researchers can reproduce the results of the same study under the same conditions. The results of the approach depend solely on the project version and the Benchmark version. Measures are taken programmatically to ensure that each project's result is deterministic and reproducible. ArDoCo's Benchmark data set is embedded in the project, which allows any researcher to reproduce the results as long as they have access to the repository and can run the program on their machine.

6 Related Work

Using informal diagrams to aid in textual Entity Recognition (ER) is a novel approach to the best of our knowledge. However, the task is related to the research domain of Software Architecture Documentation (SAD), Traceability Link Recovery (TLR), and informal diagrams in general. This section contains an overview of related work in these domains.

Ding et al. evaluate different Open Source Software (OSS) sources in a survey to determine how OSS communities use SAD. They evaluate 2,000 OSS projects and find that SAD is scarce in OSS, with only 108 projects having any SAD. The survey differentiates between *Freelance*, *Industry*, and *Research projects*. *Freelance* projects are the most common type of OSS projects, but also the least likely type to provide SAD with only 3.8%. Nonetheless, the observed format and architectural language distribution are relevant to this work. According to Ding et al., the most prevalent architectural language is natural language, with 88.9% presence. The next most prevalent languages are informal diagrams (41.7%) and Unified Modelling Language (17.6%). These three architectural languages are also popular in industry. The architectural information conveyed by the diagrams is most commonly about the model and system components. Ding et al. observe that natural language and diagrams are popular in agile OSS development because they are the simplest architectural languages. Another observation is that the most common formats for SAD in OSS are HTML and pictures. The survey affirms that the prerequisites of the approach, natural language, and informal diagrams, are prevalent in documented OSS. The survey also affirms the need for optical processing of pictures in SAD and that the perception of diagrams conveying information regarding the model and components of a system is consistent with their use in practice. This agrees with the assumptions made by this work.

Shaw and Clements [8] classify software architecture styles to formalize commonly recognized styles of problem classes. They observe that software architectures are usually expressed in diagrams consisting of lines and boxes and informal prose. They identify components and connectors as the primary constituents of architecture. According to Shaw and Clements, not every style suits every problem, but specific problems can be depicted using distinct styles. Even though informal diagram styles do not adhere to formal constraints, they appeal to common knowledge or intuition and are repeatedly used by system builders. The observations by Shad and Paul are relevant to this work because they propose that common styles are reused in informal diagrams. We can use this information to determine the styles of informal diagrams that may be of interest to our approach and to determine limitations. However, it is observed that the style of informal diagrams is often a combination of aspects from multiple styles in practice.

Jongeling et al. [16] propose an approach to turn informal diagrams into flexible blended models. They identify the lack of defined semantics and a workable representation as problems. Their approach bisects diagrams into a model part with defined syntax and semantics and a preserved remainder that can not be resolved. Transformers are used to

translate the informal diagram into this bisection or back, creating a blended modeling loop. An approach for diagram recognition is presented, where an additional legend can be supplied to define the metamodel used in the diagram. This approach was used with a script for consistency checking in an evaluative case study. A flexible blended model could be used to determine important entities in an informal diagram, which could be used to perform Diagram Text TLR as proposed by this work. However, the approach by Jongeling et al. is not fully automatic, requires capturing the diagram's metamodel, and was not evaluated quantitatively.

In a paper by Kleffmann et al. [20], TLR is used between informal diagram sketches to discover inconsistencies between sketches in collaborative sketching. They create trace links by observing the behavior of stakeholders when working with different artifacts in situ and additionally analyze extracted textual information using the Similarity Metrics (SMs) *Levenshtein distance* and *Latent Semantic Indexing*. To counter difficulties with poor handwriting, they modify the *Levenshtein distance* to be less punishing for similarly looking characters. Sketches are regarded as documents, and *Latent Semantic Indexing* is used with a query word. Results surpassing a threshold will be used to create a trace link. Inconsistencies are discovered using impact analysis. The paper's approach was evaluated in an additional paper by Kleffmann et al. [21]. *Latent Semantic Indexing* was replaced with *Vector Support Machine*, which is faster and better for small document sizes. The approach achieved a Precision of 0.93 and a Recall of 0.90. However, the trace links are between different diagrams or a diagram and an entire document. In comparison, the approach proposed by this work is not in situ. The approach also needs to create trace links between finer endpoints than informal diagrams and entire documents. The in situ approach by Kleffmann et al. uses a mix of prospective and retrospective TLR methods but our scenario is limited to retrospective methods. Kleffmann et al. propose using a modified Levenshtein distance, where homoglyphs are not counted as errors but rewarded less than an exact match. Our approach relies on multiple SMs, and not all artifacts are affected by homoglyphs. Therefore, this work proposes using a configurable character match function for flexibility instead.

Spanoudakis et al. [40] present an approach to automatically generate traceability relations between textual requirements and object models expressed in Unified Modelling Language (UML). Similarly to Architecture Documentation Consistency (ArDoCo), the TLR uses heuristic rules based on grammar to match syntactically related terms in the text to object models. This is achieved by automatically processing the text with Part Of Speech (POS) and applying traceability rules. The system calculates *belief functions* to determine its certainty in the correctness of traces and rules. In a preliminary case study, Spanoudakis et al. demonstrate that the belief functions can be used to adjust the rule deployment. In contrast to this work, the object model used in the paper adheres to the formal UML metamodel. The approach was revised in 2004 but is still limited to object models specified in UML and thus not applicable to informal diagrams [41].

7 Conclusion And Future Work

This work presented an approach to improve the Entity Recognition (ER) in Software Architecture Documentation (SAD). The SAD artifact contains Natural Language Software Architecture Documentation (NLSAD), which is often accompanied by informal diagrams. Informal diagrams consist of geometric shapes and text and do not adhere to a formal metamodel. The Entity Recognition in Software Documentation Using Trace Links to Informal Diagrams (ERID) approach uses the informal diagrams to perform a Traceability Link Recovery (TLR) between candidates for textual entities and diagram elements. ERID adds the two stages Diagram Connection Generator and Diagram Inconsistency Checker to the Architecture Documentation Consistency (ArDoCo) framework. The Diagram Connection Generator stage uses the result of the Diagram Recognition stage provided by Linking Sketches and Software Architecture (LiSSA) to perform the TLR. The trace links are subsequently used in the Diagram Inconsistency Checker stage to adjust the confidence in candidates for textual entities. The adjustment is based on heuristics and depends on the diagram coverage. The confidence reward for being covered is inversely proportional to the number of covered candidates. The confidence punishment is proportional to the number of covered candidates. Changes are performed outside of the ERID stages to facilitate the approach. A process is integrated into the Diagram Recognition to calculate references for diagram elements that can be used for similarity-based comparisons. A global abbreviation disambiguation is implemented to allow resolving abbreviations that may occur due to space constraints or convenience in both the model and informal diagrams. A good F_1-score of 0.54 is achieved for the Diagram Text TLR using the automatic diagram element extraction from images of informal diagrams.

The approach can identify and punish candidates of textual entities not contained in the diagrams. However, ERID has the limitation that it can not differentiate between the common use of a word and the use as a named entity. Another limitation is that the approach can only eliminate computer- and software-related terminology from text boxes but not supplemental text in general. This is a problem because supplemental text can lead to the creation of false diagram element references. The approach is based on the assumption that the informal diagrams contain information about entities described by the text. Nonetheless, the approach proves effective across a data set containing seven project versions spanning four projects of different architecture styles and project sizes.

The approach can be useful for tasks requiring Named Entity Recognition (NER) in a text accompanied by informal diagrams. This is shown by the integration of the approach into ArDoCo to reduce the number of falsely identified textual entities. ERID is inserted after the model and text are processed separately and candidate textual entities are created. The approach then adjusts the confidences of the candidate textual entities, which serves as a prefiltering step to the inconsistency detection based on the textual entities and linkage information. This improves Precision (P) from 0.39 to 0.70 with a negligible 0.67 to 0.64

decrease to Recall. F_1 (0.36⟶0.56), Accuracy (0.79⟶0.94) and Specificity (0.79⟶0.96) also benefit from this.

The approach is affected by the optical processing of informal diagrams. In the evaluation of the Diagram Text TLR, a worsening of F_1 from 0.74 to 0.54 was observed when relying on the automatic optical processing. A qualitative assessment pointed to issues pertaining to misread labels, complex nested hierarchies, and the process of creating diagram element references. The robustness of the approach needs to be further investigated to determine whether these issues can be mitigated independent of the optical processing. If this is not the case, future work could include improvements to the Object Detection (OD) and Optical Character Recognition (OCR) used in the Diagram Recognition. ArDoCo is currently being extended to include *Code Traceability*. Therefore, the impact of the approach on tasks such as SAD Code TLR and SAD SAM Code TLR could be the subject of future work. Currently, the approach considers neither the shape nor the color of informal diagrams. This information could be used to develop heuristics considering the relevancy of a particular diagram element or text box in future work. For example, if a diagram element contains multiple text boxes and a text box has low contrast relative to its background color, the text from the text box may be less relevant. Another aspect that could be considered is the connections between diagram elements and positional information. Currently, only parent-child relationships are considered. Additionally, the approach primarily impacts the confidence of candidate textual entities. Integrating the information from informal diagrams into the clustering process could improve the creation of candidate textual entities. The information is already present in the form of diagram-backed noun mappings but is not used when they are further clustered as of now.

Bibliography

[1] Haldun Akoglu. "User's guide to correlation coefficients". In: *Turkish Journal of Emergency Medicine* 18.3 (Sept. 1, 2018), pp. 91–93. ISSN: 2452-2473. DOI: 10.1016/j.tjem.2018.08.001. URL: https://www.sciencedirect.com/science/article/pii/S2452247318302164 (visited on 07/28/2023).

[2] Victor R. Basili and David M. Weiss. "A Methodology for Collecting Valid Software Engineering Data". In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984). Conference Name: IEEE Transactions on Software Engineering, pp. 728–738. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010301.

[3] Cambridge University Press. *proper noun*. In: *Cambridge Academic Content Dictionary*. Cambridge University Press, Apr. 26, 2023. URL: https://dictionary.cambridge.org/dictionary/english/proper-noun (visited on 04/26/2023).

[4] Deepti Chopra, Nisheeth Joshi, and Iti Mathur. "Named Entity Recognition in Hindi Using Hidden Markov Model". In: *2016 Second International Conference on Computational Intelligence & Communication Technology (CICT)*. 2016 Second International Conference on Computational Intelligence & Communication Technology (CICT). Feb. 2016, pp. 581–586. DOI: 10.1109/CICT.2016.121.

[5] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, eds. *Software and Systems Traceability*. London: Springer London, 2012. ISBN: 978-1-4471-2238-8. DOI: 10.1007/978-1-4471-2239-5. URL: https://link.springer.com/10.1007/978-1-4471-2239-5 (visited on 05/19/2023).

[6] Ernest C. Davenport and Nader A. El-Sanhurry. "Phi/Phimax: Review and Synthesis". In: *Educational and Psychological Measurement* 51.4 (Dec. 1, 1991). Publisher: SAGE Publications Inc, pp. 821–828. ISSN: 0013-1644. DOI: 10.1177/001316449105100403. URL: https://doi.org/10.1177/001316449105100403 (visited on 09/06/2023).

[7] Mark Davis and Michel Suignard. *UTS #39: Unicode Security Mechanisms*. 2022. URL: https://www.unicode.org/reports/tr39/#confusables (visited on 05/18/2023).

[8] Wei Ding et al. "How Do Open Source Communities Document Software Architecture: An Exploratory Survey". In: *2014 19th International Conference on Engineering of Complex Computer Systems*. 2014 19th International Conference on Engineering of Complex Computer Systems. Aug. 2014, pp. 136–145. DOI: 10.1109/ICECCS.2014.26. URL: https://ieeexplore.ieee.org/abstract/document/6923128 (visited on 09/30/2023).

[9] Robert Feldt and Ana Magazinius. "Validity threats in empirical software engineering research-an initial survey." In: *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering (SEKE'2010)*. 2010, pp. 374–379.

[10] Dominik Fuchß. *LiSSA/Core: v0.3.14*. July 12, 2023. URL: https://github.com/LiSSA-Approach/LiSSA-Core (visited on 10/04/2023).

[11] Dominik Fuchß et al. *ArDoCo/Benchmark*. Version ecsa22-msr4sa. Aug. 5, 2022. DOI: 10.5281/zenodo.6966832. URL: https://zenodo.org/record/6966832 (visited on 07/23/2023).

[12] Dominik Fuchß et al. "Establishing a Benchmark Dataset for Traceability Link Recovery Between Software Architecture Documentation and Models". In: *Software Architecture. ECSA 2022 Tracks and Workshops*. Ed. by Thais Batista et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2023, pp. 455–464. ISBN: 978-3-031-36889-9. DOI: 10.1007/978-3-031-36889-9_30.

[13] Wael H.Gomaa and Aly A. Fahmy. "A Survey of Text Similarity Approaches". In: *International Journal of Computer Applications* 68.13 (Apr. 18, 2013), pp. 13–18. ISSN: 09758887. DOI: 10.5120/11638-7118. URL: http://research.ijcaonline.org/volume68/number13/pxc3887118.pdf (visited on 05/06/2023).

[14] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. "Advancing candidate link generation for requirements tracing: the study of methods". In: *IEEE Transactions on Software Engineering* 32.1 (Jan. 2006). Conference Name: IEEE Transactions on Software Engineering, pp. 4–19. ISSN: 1939-3520. DOI: 10.1109/TSE.2006.3.

[15] Matthew A. Jaro. "Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida". In: *Journal of the American Statistical Association* 84.406 (1989). Publisher: [American Statistical Association, Taylor & Francis, Ltd.], pp. 414–420. ISSN: 0162-1459. DOI: 10.2307/2289924. URL: https://www.jstor.org/stable/2289924 (visited on 05/07/2023).

[16] Robbert Jongeling et al. "From Informal Architecture Diagrams to Flexible Blended Models". In: *Software Architecture*. Ed. by Ilias Gerostathopoulos et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 143–158. ISBN: 978-3-031-16697-6. DOI: 10.1007/978-3-031-16697-6_10.

[17] Jan Keim et al. "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery". In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 2023 IEEE 20th International Conference on Software Architecture (ICSA). Mar. 2023, pp. 141–152. DOI: 10.1109/ICSA56044.2023.00021.

[18] Jan Keim et al. "Trace Link Recovery for Software Architecture Documentation". In: *Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings. Ed.: S. Biffl.* 15th European Conference on Software Architecture. ISSN: 0302-9743, 1611-3349. 2021, p. 101. ISBN: 978-1-00-013839-9. DOI: 10.1007/978-3-030-86044-8_7. URL: https://publikationen.bibliothek.kit.edu/1000138399 (visited on 04/12/2023).

[19] Jóakim von Kistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. ISSN: 2375-0227. Sept. 2018, pp. 223–236. DOI: 10.1109/MASCOTS.2018.00030.

[20] Markus Kleffmann et al. “Establishing and Navigating Trace Links between Elements of Informal Diagram Sketches”. In: *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*. 2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability. ISSN: 2157-2194. May 2015, pp. 1–7. DOI: 10.1109/SST.2015.8.

[21] Markus Kleffmann et al. “Evaluation of a traceability approach for informal freehand sketches”. In: *Automated Software Engineering* 25.1 (Mar. 1, 2018), pp. 1–43. ISSN: 1573-7535. DOI: 10.1007/s10515-017-0221-6. URL: https://doi.org/10.1007/s10515-017-0221-6 (visited on 05/22/2023).

[22] B. Kolo. *Binary and Multiclass Classification*. Weatherford Press, 2011. ISBN: 978-1-61580-016-2. URL: https://books.google.de/books?id=XTuTAgAAQBAJ.

[23] Grzegorz Kondrak. “N-Gram Similarity and Distance”. In: *String Processing and Information Retrieval*. Ed. by Mariano Consens and Gonzalo Navarro. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 115–126. ISBN: 978-3-540-32241-2. DOI: 10.1007/11575832_13.

[24] Jing Li et al. “A Survey on Deep Learning for Named Entity Recognition”. In: *IEEE Transactions on Knowledge and Data Engineering* 34.1 (Jan. 2022). Conference Name: IEEE Transactions on Knowledge and Data Engineering, pp. 50–70. ISSN: 1558-2191. DOI: 10.1109/TKDE.2020.2981314.

[25] Xing Liu, Huiqin Chen, and Wangui Xia. “Overview of Named Entity Recognition”. In: *Journal of Contemporary Educational Research* 6.5 (May 30, 2022). Number: 5, pp. 65–68. ISSN: 2208-8474. DOI: 10.26689/jcer.v6i5.3958. URL: http://ojs.bbwpublisher.com/index.php/JCER/article/view/3958 (visited on 04/13/2023).

[26] Alireza Mansouri, Lilly Suriani Affendey, and Ali Mamat. “Named Entity Recognition Approaches”. In: *International Journal Of Computer Science And Network Security* 8.2 (2008), pp. 339–344.

[27] B. W. Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. In: *Biochimica et Biophysica Acta (BBA) - Protein Structure* 405.2 (Oct. 20, 1975), pp. 442–451. ISSN: 0005-2795. DOI: 10.1016/0005-2795(75)90109-9. URL: https://www.sciencedirect.com/science/article/pii/0005279575901099 (visited on 09/06/2023).

[28] Nenad Medvidovic and Richard N. Taylor. “Software architecture: foundations, theory, and practice”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE ’10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 471–472. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810435. URL: https://dl.acm.org/doi/10.1145/1810295.1810435 (visited on 04/25/2023).

[29] Antonio Menditto, Marina Patriarca, and Bertil Magnusson. “Understanding the meaning of accuracy, trueness and precision”. In: *Accreditation and Quality Assurance* 12.1 (Jan. 1, 2007), pp. 45–47. ISSN: 1432-0517. DOI: 10.1007/s00769-006-0191-z. URL: https://doi.org/10.1007/s00769-006-0191-z (visited on 09/06/2023).

[30] Ivan Mistrik et al. "Preface". In: *Relating System Quality and Software Architecture*. Ed. by Ivan Mistrik et al. Boston: Morgan Kaufmann, Jan. 1, 2014, pp. xxxi–xxxix. ISBN: 978-0-12-417009-4. DOI: 10.1016/B978-0-12-417009-4.09989-0. URL: https://www.sciencedirect.com/science/article/pii/B9780124170094099890 (visited on 04/25/2023).

[31] David Nadeau and Satoshi Sekine. "A Survey of Named Entity Recognition and Classification". In: *Lingvisticae Investigationes* 30 (Aug. 15, 2007). DOI: 10.1075/li.30.1.03nad.

[32] Roberto Navigli and Federico Martelli. "An overview of word and sense similarity". In: *Natural Language Engineering* 25.6 (Nov. 2019). Publisher: Cambridge University Press, pp. 693–714. ISSN: 1351-3249, 1469-8110. DOI: 10.1017/S1351324919000305.

[33] Nita Patil, Ajay Patil, and B. V. Pawar. "Named Entity Recognition using Conditional Random Fields". In: *Procedia Computer Science*. International Conference on Computational Intelligence and Data Science 167 (Jan. 1, 2020), pp. 1181–1188. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.03.431. URL: https://www.sciencedirect.com/science/article/pii/S1877050920308978 (visited on 04/26/2023).

[34] Princeton University. *Frequently Asked Questions*. WordNet. 2010. URL: https://wordnet.princeton.edu/frequently-asked-questions (visited on 05/15/2023).

[35] Hamid Rezatofighi et al. *Generalized Intersection over Union: A Metric and A Loss for Bounding Box Regression*. Apr. 14, 2019. DOI: 10.48550/arXiv.1902.09630. arXiv: 1902.09630[cs]. URL: http://arxiv.org/abs/1902.09630 (visited on 10/04/2023).

[36] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (Apr. 1, 2009), pp. 131–164. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: https://doi.org/10.1007/s10664-008-9102-8 (visited on 09/29/2023).

[37] Yutaka Sasaki. "The truth of the F-measure". In: *Teach Tutor Mater* (Jan. 1, 2007). URL: https://www.researchgate.net/publication/268185911_The_truth_of_the_F-measure (visited on 10/04/2023).

[38] Aaron Schlutter and Andreas Vogelsang. "Trace Link Recovery using Semantic Relation Graphs and Spreading Activation". In: *2020 IEEE 28th International Requirements Engineering Conference (RE)*. 2020 IEEE 28th International Requirements Engineering Conference (RE). ISSN: 2332-6441. Aug. 2020, pp. 20–31. DOI: 10.1109/RE48521.2020.00015.

[39] Sophie Schulz. *Linking Software Architecture Documentation and Models*. 2020. DOI: 10.5445/IR/1000126194. URL: https://publikationen.bibliothek.kit.edu/1000126194 (visited on 04/12/2023).

[40] George Spanoudakis. "Plausible and adaptive requirement traceability structures". In: *Proceedings of the 14th international conference on Software engineering and knowledge engineering*. SEKE '02. New York, NY, USA: Association for Computing Machinery, July 15, 2002, pp. 135–142. ISBN: 978-1-58113-556-5. DOI: 10.1145/568760.568786. URL: https://dl.acm.org/doi/10.1145/568760.568786 (visited on 05/26/2023).

[41] George Spanoudakis et al. “Rule-based generation of requirements traceability relations”. In: *Journal of Systems and Software* 72.2 (July 1, 2004), pp. 105–127. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(03)00242-5. URL: https://www.sciencedirect.com/science/article/pii/S0164121203002425 (visited on 05/26/2023).

[42] Yuan Tian, David Lo, and Julia Lawall. “SEWordSim: software-specific word similarity database”. In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ICSE Companion 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 568–571. ISBN: 978-1-4503-2768-8. DOI: 10.1145/2591062.2591071. URL: https://dl.acm.org/doi/10.1145/2591062.2591071 (visited on 05/07/2023).

[43] Cindy Tofthagen. “Threats to Validity in Retrospective Studies”. In: *Journal of the Advanced Practitioner in Oncology* 3.3 (2012), pp. 181–183. ISSN: 2150-0878. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4093311/ (visited on 09/28/2023).

[44] William E. Winkler. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. ERIC Number: ED325505. 1990. URL: https://eric.ed.gov/?id=ED325505 (visited on 05/07/2023).