Trace Link Recovery for Software Architecture Documentation

Jan Keim[0000−0002−8899−7081]1, Sophie Schulz[0000−0002−1531−2977]1,3, Dominik Fuchß[0000−0001−6410−6769]1, Claudius Kocher[2], Janek Speit[2], and Anne Koziolek[0000−0002−1593−3394]1

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{jan.keim, sophie.schulz, dominik.fuchss, koziolek}@kit.edu
[2] {claudius.kocher, janek.speit}@student.kit.edu
[3] Competence Center for Applied Security Technology (KASTEL), Karlsruhe, Germany

Abstract. Software Architecture Documentation often consists of different artifacts. On the one hand, there is informal textual documentation. On the other hand, there are formal models of the system. Finding related information in multiple artifacts with different level of formality is often not easy. Therefore, trace links between these can help to understand the system. In this paper, we propose an extendable, agent-based framework for creating trace links between textual software architecture documentation and models. Our framework SWATTR offers different pipeline stages to extract text and model information, identify elements in text, and connect these elements to model elements. In each stage, multiple agents can be used to capture necessary information to automatically create trace links. We evaluate the performance of our approach with three case studies and compare our results to baseline approaches. The results for our approach are good to excellent with a weighted average F_1-Score of 0.72 over all case studies. Moreover, our approach outperforms the baseline approaches on non-weighted average by at least 0.24 (weighted 0.31).

Keywords: Software Architecture Documentation · Trace Link Recovery · Modeling · Natural Language Processing · Information Retrieval.

1 Introduction

The success of a software system is highly dependent on its architecture and the architecture inhibits or enables many of the system's quality attributes [1]. Software architecture defines early design decisions about a system's remaining development, its deployment, and its maintenance. Documenting the architecture is important to capture necessary information for these tasks.

Software Architecture Documentation (SAD) is currently created in two fashions. On the one hand, there are formal models of the system. Modeling systems helps the architect to track changes in a software architecture and brings additional benefits like early evaluation and simulation to predict performance and other quality attributes [15]. On the other hand, there is informal SAD. In textual informal SAD, knowledge about the system including underlying design decisions are captured. Both are key factors to understand a system and provide essential insight.

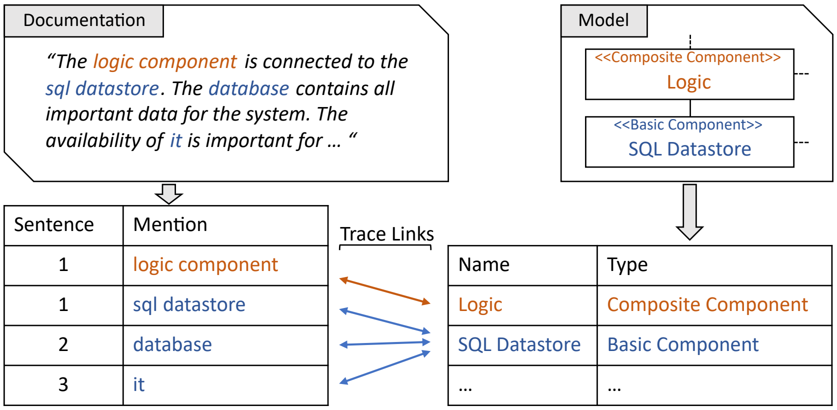
However, combining information from both artifact types is challenging. Finding information to a certain part of a system within informal documentation is not always easy, especially in large-scale systems. Vice versa, finding the model realizations from a part of the informal documentation can be time consuming. Therefore, connecting different artifacts using trace links is useful. For example, a part of the formal architecture model like a component changed. As a consequence, you need to find and update information in the informal documentation. However, a common occurrence is different naming in different artifacts [21], therefore, simple search strategies (like string comparisons) do not work. If no explicit trace links exist, identifying all locations that need to be updated can be challenging and time consuming. Additionally, trace links can help finding (in-) consistencies between artifacts, one of our long-term goals (cf. [6]).

In our work, we want to automate trace link recovery between informal textual documentation and formal models. Usually, automated trace link recovery is done for requirements to code and, to the best of our knowledge, has not been done for this case, yet. There are major differences in available information. Compared to architectural models, source code contains more explicit information, e.g., code comments and method bodies that can be used in information retrieval (IR) and natural language understanding (NLU) approaches. As a result, many approaches cannot be applied easily and due to lack of information will yield worse results. Therefore, we want to close this gap.

We propose the framework SoftWare Architecture Text Trace link Recovery (SWATTR) for creating trace links between textual informal SAD and formal models. Trace links are created between elements from different artifacts; for example, in Figure 1, where the *logic component* is realized in the model and mentioned within the text. Trace links are also created between elements where naming is only similar but not exact like between *database* and *SQL Datastore*. Within our approach, we use NLU and IR techniques to identify elements, match these elements and create these trace links.

The SWATTR framework consists of different stages: Text and Model Extraction, Element Identification, and Element Connection. Each stage uses agents to enable extensibility. This way, the framework supports adding further approaches to improve NLU and the recovery of trace links. Furthermore, the framework can be easily extended for further tasks (via additional stages) such as the previously mentioned inconsistency identification.

As a result, our research question is: How accurately can we identify trace links between textual informal SAD and models with our approach? With this work, we make the following contributions: We present an approach for creating trace links between textual informal SAD and formal models. The approach is embedded in an extendable agent-based framework. We also provide our framework, the results of our experiments, as well as a reproduction package online [7].



"The logic component is connected to the sql datastore. The database contains all important data for the system. The availability of it is important for … "

Sentence	Mention
1	logic component
1	sql datastore
2	database
3	it

Name	Type
Logic	Composite Component
SQL Datastore	Basic Component
…	…

Fig. 1. Textual mentions of architectural elements are collected. Then, they are linked to their corresponding model elements.

2 Related Work

We closely relate to the automated trace link recovery community and the problem of connecting requirement documents to code.

For the task of recovering requirements-to-code trace links, there are many approaches that are based on IR. These IR techniques dominated the trace link recovery scene for more than a decade [2]. Recent approaches like the approach by Rodriguez and Carver [16] report good results by employing machine learning techniques like evolutionary approaches. Rodriguez and Carver, for example, combine two IR metrics and use the *Non-dominated Sorting Genetic Algorithm (NSGA-II)* [3] to explore and find trace links. However, there are different problems regarding these IR approaches, including polysemy of words [20]. Moreover, underlying semantics are often disregarded.

Many more recent approaches try to understand and interpret the semantics more thoroughly, using natural language processing (NLP) and NLU techniques as well as deep learning. For example, Zhang et al. [22] use synonyms and verb-object phrases. They combine these information with structural information to create a similarity score. This score is then used to recover trace links. Guo et al. [4] use word embeddings and recurrent neural networks (RNNs), Wang et al. tackle polysemy to resolve coreferences and find terms with identical meanings [20]. These approaches often use only single measures, a shortcoming that is approached by Moran et al. [12] with a Bayesian hierarchical network. However, such approaches need lots of (training) data. Mills et al. try to face this downside with an active learning approach [10].

Tang et al. present an approach that supports traceability between requirements and architecture design [19]. They define an ontology in which specifications and architectural artifacts can be defined manually. The latter are documented in a semantic wiki. Rempel and Mäder propose traceability metrics in the context of agile development to use graph-based metrics to link requirements and test cases [14]. Building on that, Molenaar et al. propose their RE4SA approach that aims to align requirements and architecture [11]. More specifically, RE4SA creates links between Epic Stories, User Stories, modules, and features.

All these approaches investigate some form of requirements-to-code trace link recovery. However, our goal is to automatically create trace links between SAD and models, which is a slightly different problem.

3 Our Approach

The main goal of our framework SWATTR is the creation of trace links between entities of formal software architecture models and their textual informal architecture documentations. For this, we have different requirements: We want to recover trace links between mentions of a model element in text to the corresponding counterpart in the model(s) (see also Figure 1). The trace link recovery should be resilient enough to cover slightly different naming, as this is a common occurrence [21]. The framework should base on a very generic metamodel to be independent of specific ones. With this, it can be applied on different kinds of models. Additionally, the framework should have a modular design to enable easy extension. All used mechanisms should be exchangeable or configurable to adapt the framework dynamically based on given contexts. Lastly, we focus, for this paper, on the creation of trace links between text elements and, on the model side, entities like components. This means, we disregard trace links that may link to relations between components, but plan to include such in the future.

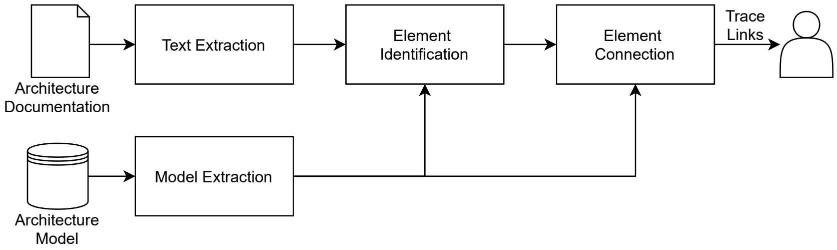


Fig. 2. The core framework consists of multiple sequential steps.

As a result, our framework SWATTR has multiple execution steps (see Figure 2) that execute different agents. The different steps are the following: *NLP Pre-Processing*, *Text Extraction*, *Model Extraction*, *Element Identification*, *Element Connection*. At first, the text is analyzed with NLP techniques. This includes, among others, part-of-speech (POS) tagging, sentence splitting, lemmatizing, and dependency parsing. Based on this, the *Text Extraction* runs to extract relevant information from the architecture documentation. Simultaneously, the *Model Extraction* extracts model elements from the architecture model. After both extraction steps, the *Element Identification* uses the extracted information from text and metamodel to identify potential elements in the text. Here, we focus on metamodel information to maintain the possibility to also identify architectural elements in the text that do not occur in the model. Lastly, the *Element Connection* connects identified elements in the text with their counterpart in the model and, thus, creates trace links.

Each step is a module that runs selected analyses. The analyses, as well as their order, are determined by configurations and can be adapted flexibly. Each analysis writes its result in a state that is held by the module, following the blackboard design pattern (cf. [9]). Additionally, every (incoming) information is associated with a confidence, so following steps can use this information.

Consequently, this design only expands promising results (dependent on the analyses) in each subsequent step. Therefore, it is important to explore many possibilities and to not discard viable solutions too early to maintain a high recall, especially in early steps such as the text extraction step. If mentions are wrongly discarded, they will not be considered for trace links. Therefore, we keep less likely options early and each step analyses and filters out very unlikely ones.

With all these measures, we try to ensure good precision and good recall.

Overall, this design leads to a modular, extendable, and exchangeable framework. In the following, we detail each of the steps within our framework.

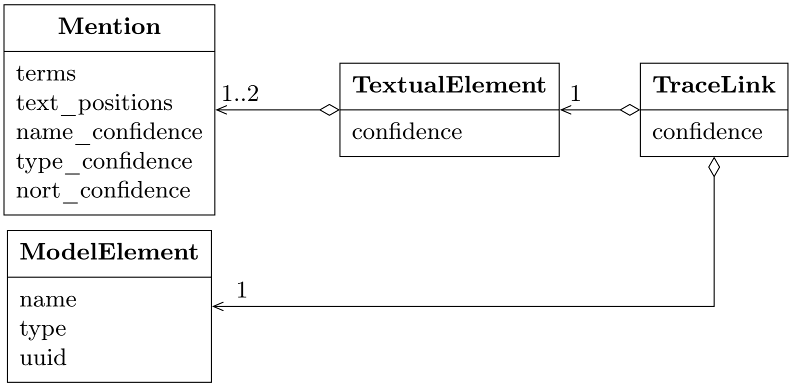


Fig. 3. Model of trace links in our approach

Extracting Text Information After initial pre-processing with NLP techniques, the text is analyzed for potential mentions. The goal of this processing step is to find out which words or compound terms denote named elements or types of elements (e.g., *component*). The mentions (cf. Figure 3) are collected in the text extraction's state. Each mention has a confidence for each classification.

Strictly looking at names and types might diminish the performance of succeeding steps, restricting the chance to identify trace links. To approach this problem, our classification approach is more lenient and also classifies mentions that could be either, a name or type (short: *nort*). Since the category *nort* is more generic, not distinct from the others, and ambiguous, *name* and *type* classifications are preferred in later steps.

For example, if an analysis classified with a 70% confidence that *database* is a *nort* and another analysis classifies that the term is with 20% confidence a *name*, the module still prefers the latter as *names* are more specific than *norts*. To ease the handling of different results, mentions also point to their most likely classification. The confidences are saved within the mention and can be used in succeeding steps, for instance, to overrule the classification.

Apart from the classification of mentions, the text extraction step clusters terms based on their word similarity. The underlying assumption here is consistent naming, thus same or similar names within the documentation are used for the same model elements. Thereby, *database* and *datastore* might occur as one entity. A similarity threshold is used to allow case-based fine-tuning for this clustering and can optionally be turned off to disable this behavior altogether.

The following enumeration describes the various analyses and heuristics that we use in this step:

Nouns – Extracts all nouns of the text and classifies them as name-or-type. If a noun occurs in plural, the noun is classified as type.

Incoming Dependencies – Examines the incoming dependencies (obtained by the dependency parser) of a word to classify it as name, type, or nort.

Outgoing Dependencies – Examines the outgoing dependencies of a word. Sources of agent and relative clause modifier dependencies are classified as norts, numeric modifier and predeterminer dependencies as type.

Pattern search – Searches for the pattern *article-type-name* in the text. If a nort follows an *article-name* combination, the nort is classified as type. If an *article-nort-type* pattern appears, the nort is classified as name.

Separators – Searches for terms that contain separators (e.g. "-"). The parts as well as the term as a whole are added to the mentions.

Compound Terms – Searches for name and type mentions. If a nort that is not yet classified as type is followed by a name, a new mention of the compound terms is created. The type detection works analogously.

Extracting Model Information The model extraction step retrieves instances from the model and saves their name, their type, and a uniquely traceable identifier (see Figure 3). These attributes can be seen as a generic definition of a model element that our approach builds upon. Exchangeable adapters can extract needed information from different metamodels and transform them into our model element definition. Right now, these attributes are sufficient for the cases we encountered. If necessary, this definition can also be extended.

The applications in our evaluation (cf. Section 4) use Palladio Component Models (PCM)[15] as architecture models. To read in models, we first transform them into an ontology using our Ecore2OWL tool[4]. We choose the ontology approach here because we plan to unify the extraction for different metamodels and we plan to extend (pre-) processing with the help of ontologies in future work. An adapter reads in the provided ontology of a PCM and extracts information to create internal representations of model elements. Additional adapters for other metamodels can be implemented similarly.

As an illustration, a PCM contains the basic component *sql datastore*. The model extractor retrieves its name, its type, and its ID using the PCM adapter. Thereby, the internal model element has the name *sql datastore* and the type *basic component* along with its id.

Identifying Elements This step uses types derived by the model extraction step and mentions from the text extraction step to identify possible elements out of mentions (from text) that should occur in the architecture model. The analyses of this step are independent from the actual model but can use metamodel information. This step also combines different identified mentions (see Figure 3) to combine names and types.

Take following sentence as example: “The logic entity gets its data from the database component”. Here, the text extractor classifies *database* as name and *component* as type and both mentions are combined. The model extractor provides the information that there is the type *basic component*. In this case, the similarity between *component* and *basic component* is close enough to identify the *database component* as potential element. Additionally, the *datastore component* is in the same cluster and, thus, is treated the same.

We use two analyses here: One builds an element whenever a *nort-type* or *type-nort* pattern occurs. The other one creates elements out of compound terms. We additionally add a copy of each element in a *name-type* combination containing only its name to avoid errors due to wrongly combined names and types. This is in line with our paradigm to keep less likely but still valid options.

Because of its independence from the actual model, the results of this step can also be used in future work within an inconsistency analysis to identify elements that are mentioned in text but are missing in the model.

Connecting Elements & Creating Trace Links In this last step, trace links are created. Here, information from textual processing as well as model processing are combined. We compare elements built out of textual mentions on one side and elements that have been extracted from a model on the other side. There are various agents that contribute to this comparison and an overall confidence for similarity is calculated. Hereby, textual elements and model instances are linked using the terms of the underlying mentions (cf. Figure 3).

A trace link is created when the comparison results in a high enough confidence (cf. Figure 3). Trace links that do not have enough confidence are discarded. Again, the minimum confidence level can be configured. Similar to previous steps, we annotate the confidence of the analysis.

[4] `https://github.com/kit-sdq/Ecore2OWL`

For example, previous steps of the approach have identified *database component* as an potential element and the *database* mention is clustered together with the *datastore*. Moreover, the model extractor retrieved a model element with the name *sql datastore* and the type *basic component*. Depending on the similarity settings, this step links the potential element to the actual model element. Thereby, all clustered occurrences of *database* and *datastore* in the text are linked to the *basic component sql datastore* in the model.

4 Evaluation

In this evaluation, we quantify the performance of our framework and answer our research question: How accurately can we identify trace links between textual SAD and models with our approach? To answer this question, we created gold standards as described in Section 4.1 for multiple case studies. We use different metrics to measure the performance; we describe these metrics in Section 4.2. Finally, we examine the results and compare them to the results of baseline approaches (see Section 4.3).

4.1 Gold Standards

To achieve our evaluation goal, the gold standard declares trace links, each consisting of the ID of a sentence in the documentation and the ID of the corresponding model element. The focus here is to have trace links when model elements are mentioned.

We use several case studies to compare the quality of our approach to others and we created a gold standard for each case study. The first case study is *Mediastore* [18], a Java EE based case study representing a (fictional) digital store to buy and download music. The second case study is *TeaStore* [8], a micro-service reference application representing a web store for tea. The third case study is *Teammates* [13], a platform for managing peer evaluations and other feedback for students.

For each of these case studies, we use existing software architecture documentation. We first remove all figures and images. We also remove semicolons to have clear separations between sentences. We use consecutive sentence numbers as ID for the textual documentation.

For all but Teammates, there was an existing Palladio model[5]. For Teammates, we reverse engineered a repository model from the code and from figures of the architecture overview from the documentation. Although the figures are close to the documentation text, there are still differences, e.g., in naming. This makes the trace link recovery problem harder and is also more realistic.

We created a trace link between a sentence and a model element if the element was mentioned in the sentence. This also includes coreferences such as “it”. Each

[5] See `https://sdqweb.ipd.kit.edu/wiki/Media_Store` and `https://github.com/ArDoCo/CaseStudies/tree/master/TeaStore`

gold standard was created by multiple authors, disagreements were discussed and resolved. One sentence can contain multiple mentions of model elements, therefore, multiple trace links per sentence can exist.

Table 1 gives an overview of the different case studies regarding their text size, number of model elements that are considered for trace links and number of trace links that are expected. Mediastore and TeaStore are both dense descriptions of the architecture with a comparably high number of trace links per sentence. Teammates is more extensive with many explanations and sentences that do not classify as a trace link.

	#Sen.	#TraceLinks	#ModelElem.	MaxTLperSen.	#Sen.w/oTL
Mediastore	37	25	14	2	13
TeaStore	43	25	13	2	22
Teammates	198	80	8	7	131

Table 1. Used case studies with their number of sentences (#Sen.), trace links (#TraceLinks), model elements (#ModelElem.), the max. number of trace links per sentence (MaxTLperSen.) & the number of sentences w/o trace links (#Sen.w/oTL)

4.2 Metrics

To quantify the results of the approaches, we use the metrics precision, recall, and F_1-Score. For this, we look at the created trace links of the approach and compare them to our gold standards. For a certain trace link, the referenced sentence id and model id must match to form a true positive case. If there is no trace link created by the approach but the gold standard expects one, we count it as false negative. If a trace link is created that is not in the gold standard, we count it as false positive. This also means, for example, if only a trace link is created between sentence 3 and the *logic component* (see Figure 1) but the gold standard expected the *sql datastore component*, then we have both, a false positive as the link does not exist in the gold standard and a false negative because an expected link is not present.

For trace link recovery in general, recall is seen as more important than precision. According to Hayes et al. [5], analysts are better at detecting false trace links than finding omitted ones. So, a result with higher recall and lower precision is preferable to one with higher precision and lower recall. However, precision has still to be regarded as the additional work for discarding false positives diminishes some of the benefits of automated trace link recovery.

4.3 Results of SWATTR

Different configurations can have a big influence on the results of our approach. We identified the threshold that determines when mentions are clustered (see

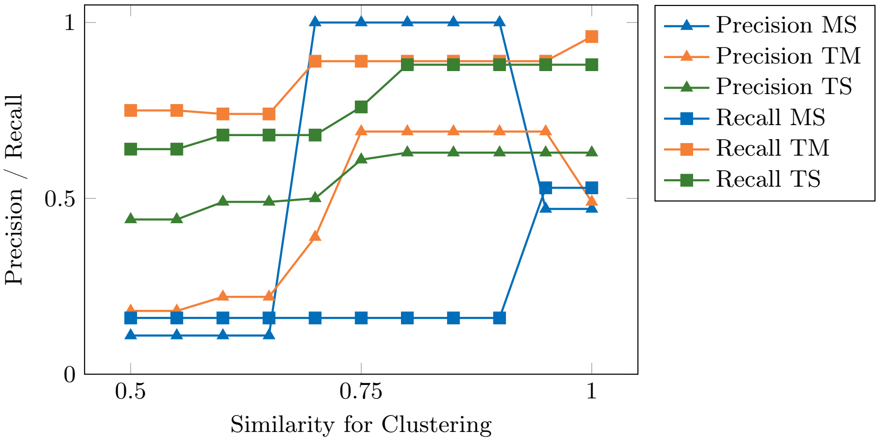


Fig. 4. Evolution of precision and recall in dependence of the similarity configuration for the clustering algorithm for Mediastore(MS), Teammates(TM), and TeaStore(TS)

Section 3) as the biggest influence factor. Therefore, we examine the performance of our approach using different thresholds. To examine the influence, we varied the threshold from 0.5 to 1.0 in steps of 0.05. The results are shown in Figure 4.

Our experiments show that recall increases with higher clustering thresholds. If the threshold is set too low, many less similar terms are clustered in few mentions. The reference terms of these mentions might not be similar enough to the names of the model elements. Thereby, trace links are not found and the recall is low. A high threshold above 0.95 results in the best recall.

Looking into the precision, best results can be achieved with threshold values between 0.8 and 0.9. With a low threshold value, many dissimilar terms are collected in a mention. This results in many incorrectly created tracelinks. If the threshold is set too high, slight differences of mentions (e.g. *database* and *datastore*) are not treated as similar anymore.

The influence of the threshold on the F_1-Score can be seen in Figure 5. Best results are achieved with thresholds between 0.8 and 0.95. The maximum F_1-Scores are 0.73 for TeaStore and 0.78 for Teammates. The performance for Mediastore is slightly worse with a F_1-Score of 0.53 at the best configuration.

According to Hayes et al. [5], the recall for TeaStore and Teammates are excellent and for Mediastore still acceptable. Precision values are good for Mediastore and excellent for TeaStore and Teammates. On average, recall values can be classified as good on the edge to excellent, precision values as excellent. For weighted average, the results can be classified as excellent.

We also looked at the source of errors in case of the best configuration. In the TeaStore case study, model elements, such as *image component*, have similar naming to general terms that were used in the text (e.g., *images*). This kind of mistake causes 46% of the false negatives. Regarding MediaStore, multiple mentions contain names of other model elements because of their similarity (e.g.

MediaManagement and *UserManagement*). Thus, every found trace link of these mentions adds at least one false positive to the result. This problem causes 69% of the false positives for this case study. Two more issues can be found in the Teammates case study. Since Teammates has a *GAE Database* but often refers to the component as well as other parts of the service as *GAE* the term is hard to distinguish. Additionally, short terms like *GAE* or *Client* can easily be too similar to other terms. The first failure causes 47% of the false positives, the latter 25%. Over all case studies, 57% of false negatives are caused by trace links that are not created due to incorrectly low similarity values. These results indicate that we should refine clustering in the future. We also need to refine our similarity calculation, especially for shorter terms.

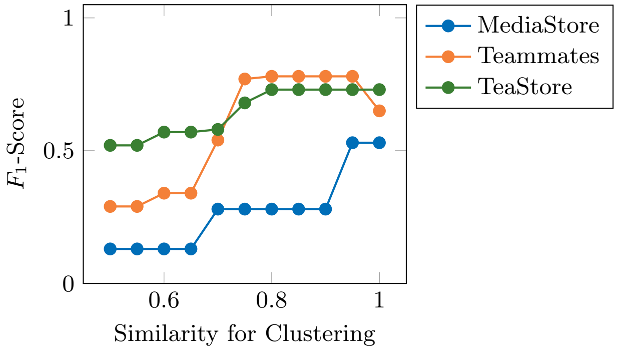


Fig. 5. Evolution of F_1-Score in dependence of the similarity configuration for the clustering algorithm for Mediastore, Teammates, and TeaStore

4.4 Comparison to other approaches

To compare our results with other approaches, we re-implemented the approaches by Rodriguez and Carver [16] and Zhang et al. [22]. We choose these approaches because they promise good (state-of-the-art) results. The second major benefit is that these approaches can be adapted to our trace link recovery problem without major problems; most needed information is still present when using models instead of code. In the following, we outline the approaches and our adaptations.

The approach as described by Rodriguez and Carver [16] uses Jaccard similarity and weighted cosine similarity and considers the trace link recovery problem as an optimization problem between these two metrics. They use the *Non-dominated Sorting Genetic Algorithm (NSGA-II)* [3] and create a population consisting of potential trace links, starting with random pairings. In each iteration, a new population is generated out of the best candidates from the previous population. Best candidates are kept and the rest of the population is created using mutation and crossover operators. As a result, the approach is highly reliant on specific configuration settings and randomness for its evolutionary aspect. We see this as a general problem of the approach, as the best configuration and best run cannot be predicted. We can mostly re-use the approach. The sole difference here is the reduced amount of text for the model side compared to code. Therefore, we expect the approach to perform worse compared to the original requirements trace link recovery problem. To cover the random factor of the approach and to identify the maximum performance of the approach, we run experiments 10 times and only select best results. Thus, results will be worse when applying this approach in a non-experimental setting.

For the second baseline approach by Zhang et al. [22], we have to adapt the described approach a little more. The approach uses synonyms, verb-object phrases, and structural information to calculate the similarity between requirements and source code. The similarity is calculated using a vector space model where the vector consists of weights for distinct words and phrases. The weights themselves are calculated based on term frequency and inverse document frequency. The terms are extracted using verb-object phrases in requirements and code and are unified by resolving synonyms. When terms on both sides are similar, a trace link is created. In contrast to the original approach, we have to adapt the verb-object phrase extraction for models and add a threshold for similarity to improve the precision. This worsens the recall slightly, so we run the approach with different threshold values and only select the best run according to the F_1-Score.

The results of our experiments for each case study are listed in Table 2. Additionally, we report two different average values. The first is the average that is calculated on the precision and recall values of the three case studies. The second one, weighted average, is the average that weights the results with the number of trace links that a case study contains. We report both values because there are slightly different semantics within these metrics. The non-weighted average mainly portraits the expected outcome for a project. The weighted average reflects the expected outcome for trace links. For example, a project with lower scores but only a small number of trace links does not affect the weighted average as much. However, both metrics have some bias. As the F_1-Score is the harmonic mean of precision and recall, we calculate the score out of the values for precision and recall, also for the average cases.

	Mediastore			TeaStore			Teammates			Average			w. Avg.		
Approach	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
SWATTR	.47	**.60**	.53	**.63**	**.88**	**.73**	**.69**	**.89**	**.78**	**.60**	**.79**	**.68**	**.64**	**.83**	**.72**
Rodriguez & C.	.07	.32	.12	.10	.20	.13	.10	.15	.12	.09	.22	.13	.10	.19	.13
Zhang et al.	**.76**	.52	**.62**	.35	.28	.31	.49	.30	.37	.53	.37	.44	.52	.34	.41

Table 2. Results of trace link recovery for the three case studies and on average (including weighted average) with comparison to baseline approaches using Precision (P), Recall (R), and F_1-Score (F_1).

Overall, SWATTR outperforms the other approaches in the case studies TeaStore and Teammates. Moreover, our framework has better results on average, both weighted and non-weighted. Only in the Mediastore case study, the approach by Zhang et al. slightly outperforms our approach, mostly due to the comparably high precision of their approach for this case study. As mentioned earlier, some model elements are very similar and more direct string comparison metrics without clustering as used by Zhang et al. can perform better here.

5 Discussion and Threats to Validity

Our approach brings a few limitations that are based on current design decisions and assumptions that we made. Currently, our approach is dependent on word-based similarity. Although there are different metrics to calculate word similarity that could be exchanged, there are still few problems: First, if one word is not recognized as similar enough to its modeled counterpart, all of its mentions in the text are ignored. This obviously has a big impact on the performance, especially recall, of our approach. Second, our approach does not differentiate different contexts of mentions yet, which might affect its precision. This limitation is also based on our aforementioned assumption that naming is consistent.

Our framework achieves promising results. Still, there are currently rather simple agents and heuristics in this prototype and further improvements need to be made. However, it is highly questionable, how much impact better and more complex agents will have, given the good results. We want to analyze the structure and properties of documentation more in depth, especially in relation to other models. This way, we want to directly tackle found properties that our approach does not address yet. We also disregard information about relations to trace relations and to find further trace links.

For most parts, the case studies in investigation use similar naming in models and documentation. Therefore, approaches that consider naming and accept slight differences can yield quite good results already. The baseline approach by Rodriguez and Carver (cf. [16]) uses such metrics, but is is affected by randomness and, thus, luck. The baseline approach by Zhang et al. (cf. [22]) also performs better when same or very similar terms are used.

In the following, we discuss the threats to validity based on the guidelines for case study research in software engineering by Runeson and Höst [17].

Construct Validity We applied commonly in the trace link recovery community used experimental designs and metrics to mitigate potential risks regarding the construct validity. However, there might be a certain bias in the selection of the use cases. We used three case studies that have different project size as well as documentation and model size. We selected different kinds of case studies with different architecture styles and patterns. This way, we believe to have reduced the bias and have a representative selection to ensure construct validity. However, these were publicly available open source systems; documentation and models of other or non-open-source projects might differ. Moreover, we only looked at component-based architecture models, which might induce bias in our experimental design. In future work, we will extend our approach and our experiments to further architecture description languages to avoid this.

Internal Validity In our case studies, we analyzed the provided documentation texts and compared it to models to create trace links. This assumes that both artifacts contain the same consistent information. Consequentially, this disregards inconsistencies between the different artifacts and further inabilities to properly map the documentation to the models. This also disregards that the abstraction level of the artifacts might be vastly different, thus a linking is less useful or applicable. We countered this factor by selecting case studies where the models and the documentation have similar abstraction and are well mappable for humans. This reduces the probability that other factors affect the investigation.

External Validity In our evaluation, we examined three different case studies. Two of which originate from research. Teammates is used in practice, but also originates from a university context. With these three case studies, we risk that not all aspects and facets of the trace link recovery problem for SAD are covered. We carefully chose the cases studies, but documentation might differ for other projects or organizations and therefore our results might differ as well. One property of these case studies is that the naming within the textual documentation is close to the naming in the models. This should be the ideal case, but inconsistent language and naming is one of the main types of inconsistencies that practitioners encounter in practice [21]. In our datasets, there is slightly inconsistent naming present, but it is unclear how representative this is overall.

Reliability For our experiments, we had to derive a gold standard for trace links in these projects ourselves. Multiple researchers each created independently a gold standard for our case studies. These gold standards where combined and the few occurring differences were discussed. This way, we tried to minimize a bias from a single researcher. However, there still can be a certain bias.

6 Conclusion and Future Work

Trace links between textual software architecture documentation and architectural models are important for tracing (in-) consistency or tracking changes. However, recovering these trace links to connect the different artifacts is not trivial and has, to the best of our knowledge, not been done, yet. In this paper, we presented the framework SWATTR for recovering trace links between these artifacts. The framework consists of multiple execution steps: Finding mentions in text, loading and analyzing provided models, identifying textual elements for potential trace links, and finally creating the actual trace links. In each step, there are different agents to provide different required analyses. This approach also allows us to flexibly add further analysis methods.

We evaluated our approach using three case studies for which we created gold standards. We calculated precision, recall, and F_1-Score for each approach. Here, our approach achieves an average F_1-Score of 0.68 (weighted 0.72) outperforming the other approaches. The average F_1-Score is 0.24 (weighted 0.31) higher than the next best baseline approach. The results of our approach are overall good to excellent according to the classification schema of Hayes et al. [5].

To overcome the identified limitations and to improve the performance of our approach, we plan to add more analyses in every step. With more specific analyses, precision as well as the recall could be increased. Especially in case of mentions, a context based disambiguation could be helpful.

We also want to look more deeply into considering and tracing relations. This can be useful, for example, to find where and how exactly the relation between two components is realized. Moreover, relations could ease the identification of trace links or help with verifying found ones.

Moreover, we want to extend our framework and combine it with other approaches. This way, we hope to improve the overall results. The most difficult research question in this regard is how to combine or select results. One basic strategy for this is to use results with most support, for example because two out of three approaches created a certain trace link. However, we also want to explore whether there are other combination strategies that can improve the overall performance of our framework.

Lastly, we want to use our framework to recognize inconsistencies between text and model. A low hanging fruit here is to classify elements in the text that could not be connected to the model as potential inconsistency.

7 Acknowledgment

This work was supported by the Competence Center for Applied Security Technology (KASTEL Project 46.23.01).

References

1. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley Professional (2003)
2. Borg, M., Runeson, P., Ardö, A.: Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability **19**(6), 1565–1616 (2014). https://doi.org/10.1007/s10664-013-9255-y
3. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation **6**(2), 182–197 (2002). https://doi.org/10.1109/4235.996017
4. Guo, J., Cheng, J., Cleland-Huang, J.: Semantically enhanced software traceability using deep learning techniques. In: 2017 IEEE/ACM 39th ICSE. pp. 3–14 (2017)
5. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: The study of methods. IEEE Transactions on Software Engineering **32**(1), 4 (2006)
6. Keim, J., Koziolek, A.: Towards consistency checking between software architecture and informal documentation. In: 2019 IEEE ICSA. pp. 250–253 (2019)

7. Keim, J., Schulz, S., Fuchß, D., Speit, J., Kocher, C., Koziolek, A.: SWATTR Reproduction Package (2021). https://doi.org/10.5281/zenodo.4730621
8. von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., Kounev, S.: Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In: IEEE 26th MASCOTS. pp. 223–236 (2018)
9. Lalanda, P.: Two complementary patterns to build multi-expert systems. In: Pattern Languages of Programs. vol. 25 (1997)
10. Mills, C., Escobar-Avila, J., Bhattacharya, A., Kondyukov, G., Chakraborty, S., Haiduc, S.: Tracing with less data: Active learning for classification-based traceability link recovery. In: 2019 IEEE ICSME. pp. 103–113 (2019)
11. Molenaar, S., Spijkman, T., Dalpiaz, F., Brinkkemper, S.: Explicit alignment of requirements and architecture in agile development. In: REFSQ 2020. pp. 169–185. Lecture Notes in Computer Science, Springer International Publishing (2020)
12. Moran, K., Palacio, D.N., Bernal-Cárdenas, C., McCrystal, D., Poshyvanyk, D., Shenefiel, C., Johnson, J.: Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In: ICSE'20. pp. 873–885. ACM (2020)
13. Rajapakse, D.C., et al.: Teammates (2021), https://teammatesv4.appspot.com
14. Rempel, P., Mäder, P.: Estimating the implementation risk of requirements in agile software development projects with traceability metrics. In: REFSQ 2015. pp. 81–97. Springer International Publishing, Cham (2015)
15. Reussner, R.H., Becker, S., Happe, J., Heinrich, R., Koziolek, A., Koziolek, H., Kramer, M., Krogmann, K.: Modeling and simulating software architectures: The Palladio approach. MIT Press (2016)
16. Rodriguez, D.V., Carver, D.L.: Multi-objective information retrieval-based NSGA-II optimization for requirements traceability recovery. In: 2020 IEEE EIT. pp. 271–280. https://doi.org/10.1109/EIT48999.2020.9208233, ISSN: 2154-0373
17. Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering **14**(2), 131. https://doi.org/10.1007/s10664-008-9102-8
18. Strittmatter, M., Kechaou, A.: The media store 3 case study system (2016). https://doi.org/10.5445/IR/1000052197
19. Tang, A., Liang, P., Clerc, V., van Vliet, H.: Traceability in the co-evolution of architectural requirements and design. In: Avgeriou, P., Grundy, J., Hall, J.G., Lago, P., Mistrík, I. (eds.) Relating Software Requirements and Architectures, pp. 35–60. Springer, https://doi.org/10.1007/978-3-642-21001-3_4
20. Wang, W., Niu, N., Liu, H., Niu, Z.: Enhancing automated requirements traceability by resolving polysemy. In: IEEE 26th RE. pp. 40–51 (2018)
21. Wohlrab, R., Eliasson, U., Pelliccione, P., Heldal, R.: Improving the consistency and usefulness of architecture descriptions: Guidelines for architects. In: 2019 IEEE ICSA. pp. 151–160 (2019). https://doi.org/10.1109/ICSA.2019.00024
22. Zhang, Y., Wan, C., Jin, B.: An empirical study on recovering requirement-to-code links. In: 17th IEEE/ACIS SNPD. pp. 121–126 (2016)