


Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery

Jan Keim 

Sophie Corallo 

Dominik Fuchß 

Anne Koziolok 

KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology
Karlsruhe, Germany

{jan.keim, sophie.corallo, dominik.fuchss, koziolok}@kit.edu

Abstract—Documenting software architecture is important for a system’s success. Software architecture documentation (SAD) makes information about the system available and eases comprehensibility. There are different forms of SADs like natural language texts and formal models with different benefits and different purposes. However, there can be inconsistent information in different SADs for the same system. Inconsistent documentation then can cause flaws in development and maintenance. To tackle this, we present an approach for inconsistency detection in natural language SAD and formal architecture models. We make use of traceability link recovery (TLR) and extend an existing approach. We utilize the results from TLR to detect unmentioned (i.e., model elements without natural language documentation) and missing model elements (i.e., described but not modeled elements). In our evaluation, we measure how the adaptations on TLR affected its performance. Moreover, we evaluate the inconsistency detection. We use a benchmark with multiple open source projects and compare the results with existing and baseline approaches. For TLR, we achieve an excellent F_1 -score of 0.81, significantly outperforming the other approaches by at least 0.24. Our approach also achieves excellent results (accuracy: 0.93) for detecting unmentioned model elements and good results for detecting missing model elements (accuracy: 0.75). These results also significantly outperform competing baselines. Although we see room for improvements, the results show that detecting inconsistencies using TLR is promising.

Index Terms—Inconsistency Detection, Traceability Link Recovery, Consistency, Documentation, Software architecture, Software engineering

I. INTRODUCTION

A system’s architecture is key for a well-engineered software system and improves the development, maintenance, and evolution of the system [1]. Preserving the knowledge about the architecture and the underlying design decisions in a software architecture documentation (SAD) further improves the benefits of a good software architecture and prevents fast deterioration [2]. According to a study by Xia et al., developers spend on average 58% of their time on comprehension [3]. Here, SAD can support developers and reduce some effort. Therefore, SAD can improve the development and the overall quality of a system.

However, one problem with SAD is the difficulty to maintain consistent documentation. This is due to the several arti-

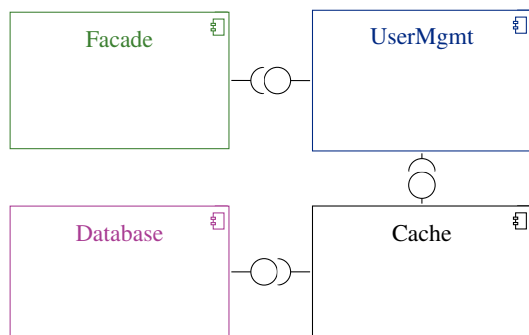
facts and types of documentation that are used in a project for different aspects. For example, there are architecture models and architecture description languages such as the Unified Modeling Language (UML) [4] and the Palladio Component Model (PCM) [5] that are used to simulate and evaluate the system at design time to assess, e.g., quality attributes (cf. [5]). Another type of SAD is informal natural language software architecture documentation (NLSAD). Due to their accessibility, NLSADs are a common choice for documentation [6]. NLSADs make knowledge about the system explicit and document design decisions. These different types of SADs are often created and updated at different times with different formality and precision [7]. This can cause inconsistencies.

According to a survey by Wohlrab et al. [8], common inconsistencies include inconsistent specifications, rules and constraints, patterns and guidelines, and wording in different artifacts. It is important to note that inconsistencies are not per se problematic [9]. For each detected inconsistency, developers can decide to tolerate or fix it. However, undetected inconsistencies do not allow such decisions and are, therefore, problematic. All benefits might perish along with important information when SAD is outdated and inconsistent [10].

In this paper, we propose our Architecture Documentation Consistency (ArDoCo) approach for inconsistency detection between NLSADs and software architecture models (SAMs). We focus on two kinds of inconsistencies: *unmentioned model elements (UMEs)* and *missing model elements (MMEs)*. We explain these in the following.

UMEs are model elements such as components that exist in the model but are not mentioned within the NLSAD. Therefore, important information such as responsibilities or underlying design decisions are not captured. For example, the *Cache* component in the example in Figure 1 is not documented in the NLSAD. Therefore, reasoning and details about this component like perceived benefits, expected workloads, planned configurations, or similar are not persisted.

MMEs are inconsistencies that occur when an NLSAD contains an architectural element that is not part of the model. In the example in Figure 1, the *Common component* from the NLSAD is missing in the model. This can happen, e.g., with



- 1) The system adheres to layered architecture.
- 2) The **Facade** is the entry point to the service.
- 3) **It** passes calls to the **user management**.
- 4) The **user management** then accesses the **DB**.
- 5) The **Common component** contains utility functionality.

Figure 1. Example architecture model and natural language documentation

prescriptive texts when something is not yet implemented in the model. In other cases, this can be a sign for outdated documentation, e.g., after refactorings.

To the best of our knowledge, detecting these kinds of inconsistencies in NLSADs and SAMs has not been tackled yet. However, based on the previous arguments, we see the need for an approach to make these inconsistencies known to developers. Our proposed approach ArDoCo locates and reports these inconsistencies and lets the user decide on ignoring, tolerating, or resolving them. The main idea is to leverage traceability link recovery (TLR) for this. TLR is generally used to create links between different artifacts, e.g., between requirements and their implementations. In our case, we look at trace links between NLSAD and SAMs. The presence of trace links shows the existence of a mutual element in both artifacts. Absent trace links can indicate inconsistencies regarding UMEs and MMEs. To do so, we build upon previous work for TLR for NLSAD named SoftWare Architecture Text Trace link Recovery (SWATTR) [11] that uses a combination of natural language processing (NLP), natural language understanding, and information retrieval. While SWATTR shows good results, one of its problems is not regarding phrases (cf. constituency parsing [12]) extensively. Therefore, we first improve TLR by adding new heuristics and by modifying existing ones to overcome this weakness. Second, we add additional processing steps to detect inconsistencies utilizing the output of SWATTR, i.e., the found trace links as well as intermediate results. For intermediate results, we use, among others, the capability of SWATTR to identify named elements in the text that should be represented in the model.

Overall, we have the following research questions:

- RQ1** To what extent do changes to the previous approach SWATTR improve the performance for TLR?
- RQ2** How does the approach perform for detecting unmentioned model elements (UMEs)?
- RQ3** How well does the approach detect missing model

elements (MMEs)?

Our main contributions with this paper are:

- We extend previous work for TLR, SWATTR [11], and add capabilities to identify inconsistencies between architecture models and NLSAD.
- We present a novel approach (ArDoCo) to identify inconsistencies regarding unmentioned and missing model elements and evaluate the approach’s capabilities.
- We provide a replication package that includes the code, baselines, our evaluation data, and results [13].

The remainder of the paper is structured as follows: In Section II, we take a look at related work and identify the research gap. In Section III, we outline the previous approach SWATTR. Section IV then introduces ArDoCo, our approach for inconsistency detection to detect unmentioned and missing model elements, including our changes to SWATTR. We evaluate the approach in Section V before we discuss the approach and the threats to validity in Section VI. Lastly, we conclude our paper in Section VII.

II. RELATED WORK

In this section, we give an overview of related work in the domain of inconsistency and its detection. There are many approaches and methods to avoid inconsistencies, for instance, by notifying users about changes on different artifacts [14]. In this section, however, we focus on approaches that detect already occurred inconsistencies between text and models.

A major area of current inconsistency detection research is between API or code documentation and implementation. The detection methods include machine learning [15], [16], static analyses [17], [18], dynamic analyses [19], and trace linking [20]. Many approaches, however, focus on pre-structured text. Mostly unrestricted textual natural language input is used by Kim and Kim [21] to detect inconsistent identifiers in API documentation and code. For their analyses, they define several kinds of inconsistencies and use NLP techniques to detect them. They collect code identifiers from trusted API documentations to map part of speech tags and synonyms to idioms, domain, and abbreviated words. Like many similar approaches, this work exploits the proximity of the text to the model. Moreover, expressions in API documentation seem to be more uniform than in NLSAD.

Another major area of inconsistency detection in software engineering deals with requirements. There are approaches on consistency between diagrams [22], between requirements and design specifications [23], or within textual requirements [24]. Fantechi and Spinicci [25] propose an approach for inconsistency detection between several textual requirements or UML class diagrams. They use part of speech tags to identify *subject-action-object* triples in the text. Similar occurrences of these subject-action-object triads indicate the possibility of an inconsistency. More precisely defined inconsistency can be found in the work of Kamalrudin et al. [26]. They support managing textual requirements and their consistency to *essential use case* models [26]. They use pattern matching

to extract interactions from textual requirements and trace them to essential use case elements. An inconsistency is detected if a phrase or element cannot be traced or has changed. The biggest differences between their approach and our approach is the type of the given SAD. Requirements are often stricter than documentation and many rules exist on how to write requirements, making them more formal. In some cases, requirements are even expressed in formal models [27]. Processing these inputs differs from our case with unrestricted, informal NLSAD. Moreover, the abstraction level between requirements and essential use case models differs from our case, making it hard to directly compare the approaches.

Inconsistency detection on architectural level is a small research area. There are approaches for inconsistency detection between automatically generated architectural decisions and *component-and-connector models* [28], between component-and-connector models and behavioral models [29], and between software specifications and implementations via architectural models [30]. These works, however, often use structured input instead of unrestricted natural language texts. During their processing, texts are regularly formalized to ease comparability. Our goal is the detection of inconsistencies between unrestricted NLSAD and SAMs. Such an approach does not yet, to the best of our knowledge, exist.

Overall, there is a number of previous work that deals with some form of inconsistency detection. However, there are few to none looking at SAM or NLSAD. Thus, we see a research gap for inconsistency detection in NLSAD, especially for unmentioned model elements and missing model elements.

III. BACKGROUND: SWATTR

Our inconsistency detection is based on TLR, specifically on SWATTR [11]. SWATTR is an extendable agent-based (i.e., heuristic-based) pipeline that takes NLSAD and several kinds of SAMs as input. The TLR pipeline of SWATTR is shown on the left side in Figure 2. There are four major processing steps, namely *model extraction*, *text extraction*, *element identification*, and *element connection*.

The *model extraction* loads model elements of a SAM into a uniform, internal representation.

In the text processing part of the pipeline, the NLSAD is first preprocessed with common NLP techniques like part-of-speech tagging, sentence splitting, lemmatizing, and dependency parsing. Based on the preprocessed text, the *text extraction* searches for names and types of possible model elements with different heuristics and agents (cf. [11] for details). Mentions of similar names or types are clustered and get a confidence value capturing the average rating of the agents for the mention being a name or a type. Thereby, in the example in Figure 1, both occurrences of *user management* would have been clustered together and treated as one mention. Since this step is part of the textual pipeline, it does not rely on any model knowledge.

In contrast, the *element identification* step combines textual and metamodel information. It uses information on existing element types from the metamodel (i.e., *Component* or *Interface*

in UML component models) to adapt the confidence of type mentions found in text based on actual model element types. Thereby in the example in Figure 1, it would become clear that *Common component* would probably be a name-type-combination as *component* is a type defined in the metamodel. The *element identification* step returns such combinations as a list of identified recommended instances (RIs). Like mentions, RIs have a confidence that represent their assumed plausibility. The confidence of an RI is combined from the confidence of the contained mentions and the rating of the agents that advocate in favor of the RI.

In the last step, the *element connection*, SWATTR creates trace links by linking RIs to model elements using word similarity metrics, namely normalized Levenshtein distance (cf. [31], [32]) and the Jaro-Winkler similarity (cf. [33]). Relations between model elements are currently not considered by SWATTR.

IV. APPROACH

As described in Section I, we build ArDoCo on a TLR approach, as this has some advantages. First, users that are interested in links between artifacts could also be interested in inconsistencies between them. One such example is estimating consequences of changing one artifact. Second and more important for us, an approach for inconsistency detection can reuse knowledge gained during TLR. Thus, our approach consists of two main processing steps, *traceability link recovery* and *inconsistency detection*, as shown in Figure 2.

For TLR, we use and adapt SWATTR [11] (cf. Section III). A benefit of SWATTR compared to other TLR approaches is that suggested model elements (RIs) can be created without any knowledge on the instantiated model. Thereby, RIs can be reused to identify mentioned model elements whose types are not contained in the model (e.g., MMEs). The found RIs are then linked to instances of the model and, thereby, the trace links are created. The inconsistency detection uses then both, RIs as intermediate results and trace links as final results of SWATTR, to locate and report inconsistent specifications (cf. [8]) in form of UMEs and MMEs. The absence of trace links for model elements corresponds to model elements that could not be found in the text is an indication for UMEs. Absent trace links for identified RIs indicate MMEs, because mentioned elements could not be found in the model. The found inconsistencies, along with existing trace links, are presented to the user to decide on their treatment.

A. Improvements to SWATTR

As the quality of our results heavily depends on the quality of the RIs and trace links of SWATTR [11], we made several adaptations to improve results. A problem with SWATTR is the rather simple handling of compound nouns that leads to inaccurate results. We thus introduce the concept of phrases (cf. [12]) and adapt heuristics to consider whether possible names and types for an RI are in the same (noun) phrase. Moreover, the previous version often recommends the software

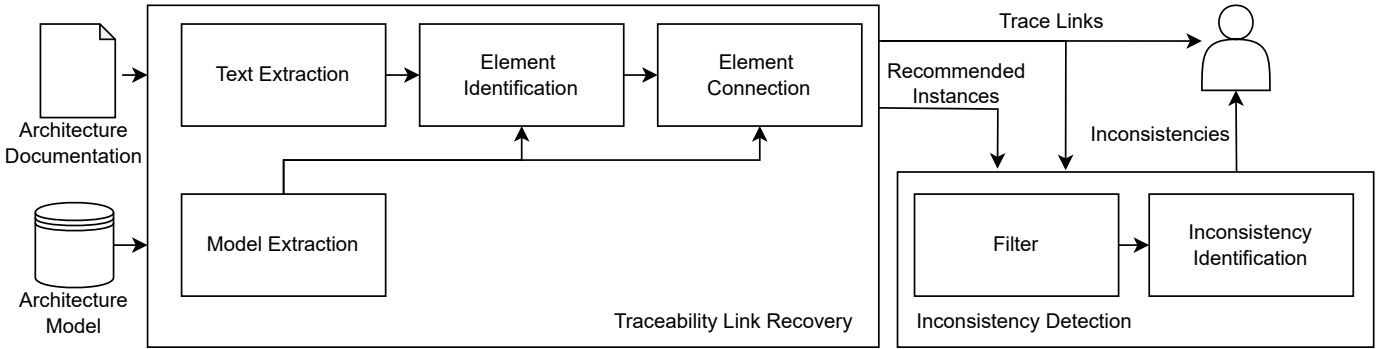


Figure 2. Overview of the approach for inconsistency detection, based on SWATTR [11]

project’s name that is irrelevant for both TLR and inconsistency detection. Thus, we ignore its mention.

Further improvements on SWATTR includes a refactoring of the similarity calculation. The approach now has a flexible, extensible design to calculate similarity using various word similarity metrics. These metrics that can be used out-of-the-box, including string-based, knowledge-based, and vector-based approaches, as well as combinations of these approaches. For this paper, we use a combination of normalized Levenshtein distance and Jaro-Winkler (cf. Section III).

We evaluate and compare both versions of SWATTR in Section V-D.

B. Unmentioned Model Elements

As described earlier, we conclude that absent trace links indicate UMEs. Unmentioned Model Elements are model elements that are not mentioned in NLSAD. One example in Figure 1 is the *Cache* component that cannot be linked to any sentence of the NLSAD. We translate this into the necessity of each model element to have at least one trace link. Thus, we base on the assumption that each model element has at least one mention in the text.

To detect UMEs between NLSAD and SAM, we first count the number of found trace links for each model element. If there are no trace links for a certain model element, the approach reports an UME, including additional information, like its name and type.

The approach allows to configure the minimum number of needed mentions (i.e., trace links) to increase the required amount of documentation that is needed for each model element. The detector then compares the number of trace links with the set threshold limit.

To add flexibility to the detection, there are further configuration options. First, users can configure preferred types from the metamodel. Thereby, only UMEs of such typed model elements (e.g., components) are investigated. Per default, we look at *Components* for UML and at *BasicComponents* and *CompositeComponents* for PCM (cf. [5]). Second, it is possible to whitelist elements that users do not want to be reported as inconsistent. For this, we support regular expression-based whitelisting. This can be handy when the user, for example,

wants to ignore inconsistencies about model elements with certain prefixes or suffixes. At the same time, a user still can provide exact names.

C. Missing Model Elements

To detect MMEs, we utilize RIs that are found in text during the processing of SWATTR. RIs represent elements in text that should also be elements in the model. Therefore, if RIs are not traced to the model (M), i.e., to at least one model element $m \in M$, there are possible inconsistencies regarding MMEs. Formally speaking, we can define it as follows:

$$\begin{aligned} \text{TLs} &\subseteq \{(r, m) \mid r \in \text{RIs}, m \in M\} \\ \text{MMEs} &= \{r_i \mid r_i \in \text{RIs} \wedge \nexists (r_i, m) \in \text{TLs}\} \end{aligned}$$

A problem lies in the general paradigm of SWATTR to not discard possible solutions too early to increase the overall recall [11]. For TLR, this is a valid strategy as the final linking step effectively filters RIs that cannot be linked. For detecting MMEs, this paradigm results in having a high number of unconnected RIs. We are looking at those unconnected RIs, including those that actually are no model elements (or very unlikely). While this allows us to achieve high recall, the precision of the approach can be quite low. Therefore, we include a filter step (cf. Figure 2) to reduce RIs and increase precision. We use different heuristics for filtering: a *threshold filter*, an *occurrence filter*, and a *filter for unwanted words*. The approach uses these filters consecutively in the above order.

The filters that we explain in the following have different configuration settings. We report default values for these settings that are based on thorough considerations and pre-studies. The user can update settings to fine-tune the approach to their use case, though.

The *threshold filter* looks at the confidence value of each RI. We use this value to filter out RIs with low confidence of being a model element, based on the heuristics (cf. Section III). However, setting the exact threshold value for a confidence value is hard and often project-dependent. To circumvent this, the filter creates dynamic threshold values based on the highest confidence value. The dynamic threshold is determined by figuring out the highest confidence value and multiplying it

with a (downsizing) factor. Based on pre-studies, we set the default factor to 0.7.

We additionally enhance the threshold filter by looking at the details within a RI regarding its name and type (cf. Section III). The idea is to sort out RIs that do not have a convincing confidence regarding their name or type. The approach determines the highest confidences for the name and type of an RI and compares them to threshold values. In the more common case, the RI needs decent confidence values (default: >0.3) for both name and type. In the other case, the RI needs a really high confidence (default: >0.8) for either a name or type. In the latter case, we assume that the RI corresponds to a model element. The threshold values for both cases are independent of the previous dynamic threshold value.

The *occurrence filter* removes RIs that seldom appear in text. The underlying assumption is that less frequent RIs are also less important. Thus, they are less likely elements that should be reflected in the model. Therefore, the approach filters all RIs that appear less than a set value, with a default value of two appearances.

Lastly, the *filter for unwanted words* looks at RIs that consist of undesired words. Often these words include certain names that are used to reference, for example, common computer science entities or terms from domain language. Since such words appear similar to actual model elements within the text, they are hard to distinguish. Therefore, we propose blacklists to filter such project- and domain-specific vocabulary. Accordingly, the approach uses two blacklists: a general blacklist and a domain-specific one.

The general blacklist contains terms that are commonly used in software engineering. These terms regularly appear in NLSAD and are similar to named model elements but are seldom used as such. Among others, the general blacklist contains programming languages (e.g., “Java”), technologies (e.g., “servlet”), and common abbreviations (e.g., “CPU”).

The project-specific blacklist allows users to set words that are commonly used within their domain and that should not represent model elements. This concerns, for example, certain technologies and tools that are used by the project, like MongoDB, Kafka, or WebRTC. Adding few words can drastically improve the performance of our tool.

In Section V-E2, we look into detail how these blacklists affect the outcome. Using such blacklists is an easy way to deal with this complex problem and improve results. However, blacklists have to be created and maintained manually. We argue that the extent of these lists as well as the effort to create them is reasonable. Users do not need to create project-specific blacklists, but can reduce the number of occurring false positives to improve the helpfulness of our approach. In future work, we plan to improve semantic analyses to omit the need for such blacklists as much as possible.

After discarding RIs based on these filters, the remaining RIs are then reported as inconsistencies. Although the approach detects these inconsistencies on a word or phrase level within a sentence, the reports of each sentence are collectively presented to the user. In our view, this makes it easier for the

user to grasp the context. Further, this eases comparisons with competing or baseline approaches (cf. Section V-A).

V. EVALUATION

In this section, we evaluate our approach for TLR and inconsistency detection to answer our research questions (cf. Section I). We use the Goal Question Metric approach [34] to structure the evaluation and assess the performance of our approach in different evaluations.

The first goal (G1) is to link sentences that mention a certain model element to the corresponding model element (TLR). This goal can be answered with RQ1 about measuring the effects of our changes in SWATTR.

The second goal (G2) and third goal (G3) deal with inconsistency detection. G2 is about finding model elements that are not described in text. This goal leads to RQ2 about the performance of detecting UMEs. G3 is about finding expected model elements in text that are missing in the model. This goal is answered with RQ3 about our capabilities to detect MMEs. Moreover, an additional question is how the different blacklists affect its performance.

We answer our research questions using different metrics: precision, recall, F_1 -score, accuracy, specificity, and Φ . We introduce these metrics in Section V-C.

Parts of the evaluation are aligned with the evaluation of the previous approach. The structure of this section is as follows: Section V-A introduces baseline approaches for TLR and inconsistency detection. In Section V-B, we present the used dataset before introducing our metrics in Section V-C. We then compare the results of the TLR approach with previous work (SWATTR) and a baseline approach in Section V-D. Lastly, we evaluate our results for inconsistency detection in Section V-E.

A. Baseline Approaches

To be able to set the evaluation results into perspective, we need comparative approaches. For TLR, we can compare our adaptations with the previous version of SWATTR [11]. We previously showed that SWATTR outperforms other TLR approaches that were adapted to work with NLSADs and SAMs [11]. Apart from that, there are, to the best of our knowledge, no approaches that look directly into the same problem. In this section, we introduce baseline approaches for the given problems. These approaches represent basic solutions that are based on certain assumptions about their respective problems. Unfortunately, there is no baseline for detecting UMEs.

1) *Traceability Link Recovery*: The baseline approach for TLR uses the assumption that elements that should be linked have equal or really similar naming. This assumption is generally used for Information Retrieval approaches. Thus, the baseline approach employs common techniques from Information Retrieval: n-grams and word similarity techniques.

We extract n-grams, specifically unigrams, bigrams, and trigrams, for the words of each sentence and for the words in each model element. The approach then compares the n-grams

from the text with the n-grams from the model. The comparison ignores casing and uses the normalized Levenshtein distance as defined by Charlet and Demnati [32] to determine the similarity of n-grams. If the similarity of two n-grams exceeds a given threshold, we create a trace link between the sentence and the model element the n-grams belong to. Based on empirical evidence while optimizing on F_1 -score, we use a threshold of 0.9. This threshold leads to precise results and reasonable recall. Reducing the threshold increases the recall slightly but is demanding towards precision.

2) *Detecting Missing Model Elements*: As there are no existing approaches for detecting MMEs, the goal of the baseline is to present a lower boundary.

The baseline approach is based on the assumption that each sentence in the textual documentation is related to a model element. We base the assumption on the foundational idea that there are no wasteful sentences in NLSAD. Thus, each sentence should have at least one associated trace link. Using this assumption, the baseline approach detects sentences that have no trace links. For these sentences, the approach reports that there are inconsistencies regarding MMEs.

The challenging part of this task is the detection of meaningful sentences w.r.t. the model. There are sentences that describe architectural properties that cannot be represented with the model. Examples include design rules or technological decisions (cf. [35], [36]) or sentences that add further explanation, set the scope, or similar. The first sentence in the running example in Figure 1 is an example for this. Additionally, the assumption disregards that one single sentence can mention multiple model elements. If one of these elements is actually missing in the model, the baseline approach does not detect the inconsistency.

The mentioned special cases are usually present in few sentences. Overall, we expect the baseline approach to achieve good recall and, thus, this baseline serves as good lower bound for the evaluation. The different results between the baseline and our approach show the capabilities of our approach in this regard. We further discuss assumptions in Section VI.

B. Case Studies

For the evaluation, we use case studies of a benchmark dataset [37] already used in previous work [11]. This benchmark is created for TLR between NLSAD and SAM (cf. [37]). It consists of NLSADs and SAMs of several open source software projects and provides gold standards for each project. The gold standards define the expected trace links between a sentence in NLSAD and specific elements in an SAM.

The benchmark’s initial dataset consists of three open source software projects *MediaStore (MS)*, *TeaStore (TS)*, and *TEAMMATES (TM)*. We additionally extend the dataset by adding the projects *JabRef (JR)* and *BigBlueButton (BBB)* to improve validity towards generalizability. BigBlueButton is a web conferencing system that contains different programming languages and different technologies. JabRef is a bibliography manager written in Java that is, alongside TEAMMATES,

Table I
PROJECTS FOR EVALUATION, BASED ON [37]

Project	Language (kLOC)	Forks	Contributors
MediaStore (MS)	Java(4)	—	—
TeaStore (TS)	Java(12)	0.1k	≈ 15
TEAMMATES (TM)	Java(91), TypeScript(54)	2.6k	≈ 500
BigBlueButton (BBB)	JavaScript(69), JSX(47), Scala(22), Java(21)	5.8k	≈ 180
JabRef (JR)	Java(157)	2.0k	≈ 490

widely used as case study in the software architecture community (cf. [38]–[40]).

We also extend this benchmark with alternative versions of NLSADs and the corresponding gold standards, where possible. These NLSADs have been taken from the past (historical) versions of the respective projects. This way, we intend to gain more insights into software evolution scenarios.

On the model side, we use the same models as in previous work (cf. [11]). For JabRef, we create models based on existing ones within the software architecture community (cf. [38]–[40]). For BigBlueButton, we manually reflect their architecture in both a PCM and UML model based on the code and an architecture overview figure in their NLSAD.

In Section VI, we discuss the creation of the gold standards and reflect on threats to validity.

Table I shows an overview of the projects used for evaluation. The table shows characteristics of the projects like the primarily used programming languages with lines of code (LOC), the number of forks, and the number of contributors. The projects have different characteristics like size or programming language. Moreover, TM and JR have a considerably higher number of contributors that can benefit from consistent documentation.

C. Metrics

For our evaluation, we use the metrics **Precision (P)**, **Recall (R)**, **F_1 -score (F_1)**, and **Accuracy (Acc)** that are commonly used in TLR and similar research areas (cf. [41], [42]). Apart from these standard metrics, we also look at **Specificity (Spec)** and the **Φ coefficient**.

With **Specificity (Spec)**, the true negative rate, we assess the probability of not falsely reporting something as trace link or inconsistency:

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

With the **Φ coefficient** [43], also known as Matthews correlation coefficient [44], we measure how the results correlate with the expected values. This allows us to quantify how well an approach differs from chance. This metric is often used in machine learning and is seen as superior to accuracy, F_1 -score, and similar metrics [45], [46]. Other than previous metrics that have values ranging between 0 and 1, the values of Φ range between -1 and +1. A Φ of +1 shows perfect correlation, -1 shows total disagreement. Random predictions achieve a value of 0. It needs to be noted that Φ usually does not achieve

values of +1 or -1. At the same time, a value that is not equal to +1 or -1 does not directly state how similar it is to random guessing [46]. To deal with this, we use the common trick to normalize Φ with the help of its maximum achievable value, Φ_{max} [47], [48]. This normalization then shows the relation to random guessing. In the following, the formulas for calculating Φ , Φ_{max} and the normalized version Φ_N are presented:

$$\begin{aligned}
 P_1 &= TP + FP & Q_1 &= TN + FN \\
 P_2 &= TP + FN & Q_2 &= TN + FP \\
 R_1 &= \sqrt{P_1 Q_2} & R_2 &= \sqrt{P_2 Q_1} \\
 \Phi &= \frac{TP * TN - FP * FN}{\sqrt{P_1 P_2 Q_1 Q_2}} \\
 \Phi_{max} &= \begin{cases} R_1/R_2 & \text{if } P_2 \geq P_1 \\ R_2/R_1 & \text{otherwise} \end{cases} \\
 \Phi_N &= \Phi/\Phi_{max}
 \end{aligned}$$

In addition to the results per project, we provide the macro average (Avg.) as well as a weighted average (w. Avg.) for every metric. We weight each project’s result by the number of expected trace links or the number of expected inconsistencies. The macro average allows us to summarize the expected results when applying our approach on a project. The weighted average instead shows the expected results per trace link or per inconsistency, i.e., how likely each instance will be detected.

D. Traceability Link Recovery

In this part of evaluation, we want to tackle G1 and RQ1. To answer RQ1, we compare the results of the approach for TLR with previous work (SWATTR) as well as the previously described baseline approach that performs TLR between NLSAD and SAM. Since previous work shows that SWATTR outperforms similar approaches [11], we focus on comparing the current approach with SWATTR and the baseline approach.

To use the metrics from Section V-C, we need to define what exactly TP, TN, FP, and FN are. We use the same definitions as in previous work [11]. Trace links consist of two parts, the sentence number and the model element. We define a TP as a reported trace link where both parts match to a trace link in the gold standard. FP are those trace links that are found but not defined in the gold standard. FN are defined as trace links that are present in gold standard but are not found by the TLR approach. Finally, TN denotes all those trace links that are correctly not reported. We calculate their number by taking into account that we link a finite set of sentences and a finite set of model elements. Therefore, we can use the number of possible combinations of sentences and model elements: $\#Sentences \times \#ModelElements = TP + FP + FN + TN$. As we have TP, FP, and FN, we can calculate TN.

Table II shows the detailed results of the approach for the different metrics. Additionally, Table III compares the macro average and weighted average of our approach with the baseline and the previous work SWATTR.

Table II
DETAILED RESULTS FOR TLR WITH (WEIGHTED) AVERAGE ((W.) AVG.) RESULTS FOR CURRENT AND HISTORICAL TEXTS AND OVERALL (Σ).

	Project	P	R	F ₁	Acc	Spec	Φ	Φ_N
Current	MS	1.0	.62	.77	.98	1.0	.78	1.0
	TS	1.0	.74	.85	.99	1.0	.85	1.0
	TM	.56	.90	.69	.97	.98	.70	.89
	BBB	.88	.83	.85	.99	.99	.84	.87
	JR	.90	1.0	.95	.97	.97	.93	1.0
	Avg.	.87	.82	.82	.98	.99	.82	.95
	w. Avg.	.83	.82	.80	.98	.99	.80	.93
Historical	TS	1.0	.93	.97	1.0	1.0	.96	1.0
	TM	.52	.70	.60	.97	.98	.59	.68
	BBB	.81	.62	.70	.98	.99	.70	.80
	JR	.82	1.0	.90	.97	.96	.89	1.0
	Avg.	.79	.81	.79	.98	.98	.79	.87
	w. Avg.	.80	.79	.79	.98	.99	.78	.86
Σ	Avg.	.83	.82	.81	.98	.99	.80	.92
	w. Avg.	.81	.81	.80	.98	.99	.79	.90

Table III
COMPARISON OF THE AVERAGE RESULTS FOR TLR OF OUR APPROACH WITH SWATTR AND THE BASELINE

Approach	P	R	F ₁	Acc	Spec	Φ	Φ_N
Average							
- Baseline	.82	.38	.51	.89	.98	.50	.79
- SWATTR	.53	.69	.57	.94	.95	.56	.70
- Ours	.83	.82	.81	.98	.99	.80	.92
Weighted Avg.							
- Baseline	.80	.37	.50	.89	.98	.49	.76
- SWATTR	.49	.63	.52	.94	.96	.52	.66
- Ours	.81	.81	.80	.98	.99	.79	.90

According to the classification scheme of Hayes et al. [41], the approach achieves *excellent* results on average for precision, recall, and F1. We want to emphasize the improvement of precision compared to the previous work [11]. Additionally, the recall for the approach is increased compared to SWATTR. For F1-score, our approach significantly outperforms the other approaches according to the one-sided Wilcoxon signed-rank test ($\alpha = 0.05$). Moreover, specificity and recall are also close to 1. Good results in these metrics are important as they affect the inconsistency detection approach. Φ and Φ_N also show that our results closely correlate to the expected results.

To summarize RQ1, the changes on SWATTR lead to improvements in all metrics. Regarding precision, recall, F1-score, and Φ_N , the changes improve results in each metric by more than 20 percentage points (pp). Respective accuracy and specificity, we achieved small improvements of 4 pp. With an F1-score of 80%, Φ_N of 0.90 and accuracy and specificity around 100%, our approach is performing excellent and is outperforming SWATTR and the baseline in all metrics.

E. Inconsistency Detection

In this part of evaluation, we address RQ2 and RQ3 by evaluating our approach w.r.t. G2 and G3. We first focus on G2, the detection of model elements that are not described

Table IV

DETAILED RESULTS FOR DETECTING UNMENTIONED MODEL ELEMENTS FOR CURRENT AND HISTORICAL TEXTS AND OVERALL (Σ). E IS THE NUMBER OF EXPECTED ELEMENTS.

	Project	E	P	R	F ₁	Acc	Spec	Φ	Φ_N
Current	MS	4	.67	1.0	.80	.88	.83	.75	1.0
	TS	5	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BBB	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	JR	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Avg.		.92	1.0	.95	.98	.97	.94	1.0
	w. Avg.		.88	1.0	.93	.95	.94	.91	1.0
Historical	TS	6	1.0	.83	.91	.91	1.0	.83	1.0
	TM	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BBB	4	.50	.75	.60	.73	.73	.43	.58
	JR	3	1.0	.67	.80	.83	1.0	.71	1.0
	Avg.		.88	.81	.83	.87	.93	.74	.90
	w. Avg.		.86	.79	.80	.85	.92	.70	.88
Σ	Avg.		.90	.91	.89	.93	.95	.84	.95
	w. Avg.		.87	.88	.86	.90	.93	.79	.93

in the text (*UMEs*), in Section V-E1. In Section V-E2, we analyze our approach for G3, the detection of described but not modeled elements (*MMEs*).

1) *Detecting Unmentioned Model Elements*: At first, we look at the results of the approach w.r.t. identifying missing textual documentation for model elements.

In order to evaluate the approach, we extend the gold standard of the benchmark dataset: The UME gold standard labels model elements as inconsistent that have no trace links according to the gold standard for TLR. Two persons have each created a gold standard and discussed their results to form the final gold standard.

In this setting, we define TP as UMEs that are correctly identified as such, according to the gold standard. In the same fashion, TN are all model elements that are correctly not classified as inconsistent. FP are model elements that are wrongly reported as UMEs. Similarly, FN are model elements that are falsely not reported as inconsistent UMEs.

Table IV shows the detailed results of the approach. We also provide the number of these expected UMEs (E) to give a better overview of the problem and subsumption of the results. The table does not contain results for TM as the gold standard does not contain any UME for it. The approach correctly reports no inconsistencies for this project. However, some metrics cannot be applied to such scenarios and, therefore, we omit this project.

The results in Table IV show that the approach achieves excellent results and detects nearly all inconsistencies regarding UMEs with high precision for most projects. The only outlier is historical BBB. In this case, some incorrect and unidentified trace links that also negatively affect TLR apply to UMEs. The results rely on good performance for TLR and are directly affected by it. In cases where our TLR approach underperforms, inconsistency detection is clearly affected.

Despite the outlier, we get excellent results for detecting UMEs. In general, the weighted average F₁-score achieves 0.86 whereas accuracy, specificity, and Φ_N range from 0.90

Table V

DETAILED RESULTS FOR DETECTING MISSING MODEL ELEMENTS FOR CURRENT AND HISTORICAL TEXTS AND OVERALL (Σ).

	Project	P	R	F ₁	Acc	Spec	Φ	Φ_N	
Current	MS	.21	.79	.33	.70	.69	.23	.68	
	TS	.96	.70	.79	.96	1.0	.81	.95	
	TM	.18	.76	.28	.85	.85	.29	.71	
	BBB	.89	.46	.43	.96	.99	.54	.65	
	JR	1.0	.44	.44	.85	1.0	.62	1.0	
	Avg.		.65	.63	.45	.86	.91	.50	.80
w. Avg.		.60	.63	.43	.87	.90	.47	.75	
Historical	TS	.16	.98	.28	.38	.29	.15	.94	
	TM	.17	.63	.26	.86	.87	.26	.57	
	BBB	.09	.18	.11	.81	.87	.02	.04	
	JR	.22	.11	.15	.57	.78	-.09	-.11	
	Avg.		.16	.48	.20	.66	.70	.09	.36
	w. Avg.		.14	.47	.19	.71	.74	.11	.36
Σ	Avg.		.43	.56	.34	.77	.82	.31	.60
	w. Avg.		.39	.64	.34	.77	.78	.32	.66

to 0.93. Thereby, we are confident to conclude excellent performance of our approach for RQ2.

2) *Detecting Missing Model Elements*: In this last part of our evaluation, we focus on RQ3. We analyze if we accomplish G3, the identification of elements in text that should be modeled but are not part of the SAM, based on RQ3 and the previously defined metrics. In addition, we tackle the question of how the blacklists for unwanted words (cf. Section IV-C) affect the performance.

We use the same projects and benchmark dataset as before. We simulate missing model elements by removing them from the model while conserving the text. The trace links defined in the gold standard then indicate inconsistencies for the removed model element. Therefore, we can utilize the existing gold standard and automate the evaluation: We conduct multiple runs for the inconsistency detection approach. In each run, we provide a SAM with all model elements except one that is removed for this run. We repeat runs until each model element was removed once. For each run, we then assess how well the approach detected the introduced inconsistency. As we perform multiple runs for each project, we accumulate the results of all runs within a project.

In this setting, TP are those inconsistencies that correctly mark an MME. FP are then those inconsistencies that are incorrectly reported as MME inconsistency. Consequently, FN are all sentences that contain an inconsistency but are not labeled as inconsistent by the approach. Finally, TN are all sentences that are correctly not marked as inconsistent.

Table V shows the detailed results of the approach with results for each project. For the results in this table, we use the common as well as project-specific blacklists. Detailed lists of these blacklists can be found in the dataset [13]. We do not use any whitelists for this evaluation.

The results are mixed depending on the project. For some projects like TS, BBB, or JR, the approach can detect inconsistencies with high precision and quite good recall. For most other projects, precision is rather low compared to recall. Still,

Table VI
INFLUENCE OF FILTERS FOR INCONSISTENCY DETECTION OF MISSING
MODEL ELEMENTS

Approach	P	R	F ₁	Acc	Spec	Φ	Φ_N
Average							
- Baseline	.09	.58	.15	.39	.37	-.02	.02
- Ours (All Filters)	.43	.56	.34	.77	.82	.31	.60
- Ours (Common)	.27	.58	.27	.71	.75	.21	.55
- Ours (No Filters)	.17	.61	.24	.65	.66	.13	.44

Φ_N indicates good results. An average recall of 0.56 shows that we find more than half of the contained inconsistencies.

When not factoring in the outliers, the recall is even higher. Outliers for this are historical BBB and JR. For both projects the difference between the versions are quite big. Additionally, the documentation consists of a comparably high amount of noise (i.e., named entities that are not directly related to the model). The approach has problems differencing noise from actual model elements. We plan to improve the approach for such cases in future work.

Table VI contains the average results for the baseline approach. Comparing the results of our approach to the baseline, we conclude that our approach outperforms the baseline in all metrics except recall when applying all filters, where we achieve similar results. Regarding F₁-score, our approach significantly outperforms the baseline approach in all settings with a significance level α of 0.05. Our approach has considerably better results for Φ_N . As expected, the baseline approach achieves high recall but is imprecise.

Table VI additionally compares the results for the different configurations regarding the blacklists for filtering unwanted words (cf. Section IV-C). *No Filters* means that we disable the unwanted words filter. *Common* refers to the global blacklist for unwanted words, containing common software engineering-related terms. *All Filters* refers to the combination of the common blacklist and a project-specific blacklist. For our case studies, the latter contains on average 10 words, ranging from 5 to 22 words (median: 7). The exact content of the blacklists can be found in the reproduction package [13]. The results show that adding filters increases the precision by a lot, whereas recall decreases only slightly.

To answer RQ3, the results are overall promising and show the capabilities of the approach. In general, with all filters enabled, we achieve a weighted average F₁-score of 34% whereas accuracy, specificity, and Φ_N lie between 0.60 and 0.82. Even though the results seem to be promising, there is still room for improvements, especially to better filter false positives and to avoid outliers.

VI. DISCUSSION

Our approach for TLR and inconsistency detection has limitations that are based on assumptions and design decisions. As the inconsistency detection builds on SWATTR, we inherit most of these limitations [11].

We assume that software architecture documentation is written in English. In order to adapt SWATTR to other languages,

the pre-processing and many heuristics for TLR have to be exchanged. However, the inconsistency detection is mostly language-independent.

Additionally, we assume that the text refers to model elements and uses similar and unique naming. With dissimilar names, the approach fails to recover trace links and falsely identifies inconsistencies. Ambiguous names also threaten TLR and consequently inconsistency detection. To mitigate this risk, the approach needs further improvements, e.g., by considering contexts from both textual and model side. This would require the tracing of relations. Nevertheless, the approach leads to promising results and shows great potential for these future improvements.

Finally, we have an assumption regarding inconsistency detection. We assume that each sentence in an NLSAD should be traceable to some model element of an SAM. The assumption reflects the common decision in TLR that each trace artifact is traced (cf. [41], [42], [49]). Automated approaches typically select the best option(s) for each trace artifact in the source artifact type, e.g., the best class(es) for a requirement. As our inconsistency detection heavily depends on trace links between sentences and model elements, we adopt the common practices of TLR to our case. The assumption is slightly overestimating the importance of each sentence, but is a good approximation for its objective.

In the following, we discuss the threats to validity based on the guidelines for case study research in software engineering by Runeson and Höst [50].

a) *Construct Validity*: We use common TLR experimental designs and metrics for our evaluation to mitigate risks regarding construct validity. The results rely on case studies that are already used in literature. The cases cover different project and model sizes, architecture styles, and patterns. This reduces bias when choosing a representative collection.

b) *Internal Validity*: In our approach, we assume that SADs are consistent. In our case, this also implies that NLSAD and SAM are on similar levels of abstraction. We thereby disregard trace links between vastly different abstraction levels or links between more than two elements. Whether such trace links can be useful and whether they are applicable to our use cases is part of future research.

Although we can create fine-grained trace links on word- or phrase-level, we report trace links on sentence-level. We choose this output as it is common practice in the TLR community and allows us to compare our approach against others. However, this has some side effects. First, a correct trace link can be found even though the approach does not identify the exact mention in a sentence. Second, only one trace link for a given model element and sentence is effectively detected. Since our inconsistency detection builds on SWATTR's results, it is influenced by these effects.

For the evaluation of inconsistency detection, we searched for varying versions of NLSADs and SAMs. However, we only found alternating (historical) NLSADs in four of the five cases. Moreover, we could barely find older models. Thus, we assume the varying versions to be consistent with the current model,

which is not necessarily given. For the detection of MMEs, we decided to create artificial inconsistencies by removing model elements to simulate them. These artificial inconsistencies are possibly not always realistic. For the evaluation of UME detection, we compare the historical documentations to the current model. This combination takes into account the realistic evolution of artifacts. Although the scope of the changes between the two versions could be criticized to be larger than between two usual consecutive versions, we expect that this reduces the bias mentioned above.

c) External Validity: We carefully selected the case studies from open source projects. Since the artifacts from such projects, especially SADs, might differ from closed source projects, this could affect generalizability. We selected only case studies with component-based architecture models. In future work, we want to extend the experiments to other architecture models to mitigate this bias. Additionally, two of the case studies have their origin in academia. To balance this, we used three non-academic ones. Lastly, with a total of only five case studies, we risk to cover not all aspects of trace links and inconsistencies in our evaluation. Therefore, this collection should be extended in future work.

d) Reliability: For this paper, we reused and extended a benchmark dataset. However, we updated the gold standards as we identified some mistakes and wrong definitions in the previous version. For instance, packages that shared their name with a component were treated as mentions of referred components. We argue that there is a difference between talking about package names and components and, therefore, we do not regard these cases as trace links. This affects the comparability between this and the previous work. In order to reduce bias, multiple researchers (re-) created the gold standards independently. Afterwards, they discussed their differences as well as differences to the existing gold standards. This way, we improve correctness and reduce bias of the gold standards. We then applied SWATTR to the updated gold standards to compare its results. Thus, we believe to have reduced the impact of the revised gold standards.

VII. CONCLUSION

In this paper, we looked into the automatic detection of inconsistencies in software architecture documentations (SADs). SADs can contribute to successful software development and are thereby beneficial. However, undetected inconsistencies in SAD, like inconsistent specifications, can introduce problems, lead to misunderstandings, and threaten development.

To uncover inconsistencies, we proposed our ArDoCo approach for inconsistency detection by combining traceability link recovery (TLR) with further heuristics. For this, we extended previous work in TLR between natural language software architecture documentation (NLSAD) and software architecture models (SAMs). We adapted and extended the SoftWare Architecture Text Trace link Recovery (SWATTR) approach for TLR. Our approach detects inconsistencies regarding unmentioned model elements (UMEs) and missing model elements (MMEs). UMEs are found via not traced

model elements whereas MMEs are identified with absent trace links for perceived model element in NLSAD.

The TLR and inconsistency detection steps of our approach are individually evaluated using a benchmark of five open source projects. We compare our approach with existing and baseline approaches. The results show that our approach significantly outperforms the other approaches ($\alpha = 0.05$).

For TLR, the approach achieves an average F_1 -score of 0.81 and an accuracy of 0.98. We achieve a minimum increase in F_1 -score of 0.24 and in accuracy of 0.09 compared to competing approaches. The Φ -metric also indicates that the results of our approach correlate closely with expectation.

The detection of UMEs achieves an average F_1 -score of 0.89 and an accuracy of 0.93. Lastly, the evaluation for detecting MMEs achieves an average F_1 -score of up to 0.34 and an average accuracy of 0.77. The results of our approach are significantly better than the results of the baseline approach.

Overall, we showed that using trace links to detect inconsistencies is reasonable and promising. We can use the absence of trace links to identify and pinpoint inconsistencies. In this paper, we focused on UMEs and MMEs but expect to be able to use this approach for further kinds of inconsistencies.

Despite promising results, we see room for improvements.

Regarding the processing steps, we want to investigate whether we can improve the identification of mentioned model elements in NLSADs. The underlying problem can be seen as a special variant of the named entity recognition problem in natural language processing. State-of-the-art approaches for named entity recognition use deep learning and neural language models (cf. [51]–[53]). However, our mention detection searches for project-specific entities whereas most named entity recognition approaches base on widely used entities. We plan to investigate whether we can utilize transfer-learning for our special use case. The biggest hurdle for using such approaches is the lack of training data.

The detection of mentions, trace links, and inconsistencies could also profit from a precise identification of sentences with model-related design decisions. In future research, we want to ignore unrelated sentences by classifying sentences based on contained design decisions to increase precision.

Lastly, relations can help to identify the context of a model element, similar to design decisions. Concerning relations could ease the detection of mentions and thereby improve TLR and inconsistency detection. However, relations can also be inconsistent between NLSAD and SAMs. Therefore, we want to investigate whether we can extend our approach to tracing relations and identifying inconsistent relations.

ACKNOWLEDGMENT

This work was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. This publication is also based on the research project SofDCar (19S21002), which is funded by the German Federal Ministry for Economic Affairs and Climate Action.

REFERENCES

- [1] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 471–472.
- [2] D. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*, 1994, pp. 279–287.
- [3] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018.
- [4] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert, "Unified modeling language (UML) version 2.5.1," Object Management Group (OMG), Standard, Dec. 2017. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1>
- [5] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolok, H. Koziolok, M. Kramer, and K. Krogmann, *Modeling and simulating software architectures: The Palladio approach*. MIT Press, 2016.
- [6] W. Ding, P. Liang, A. Tang, H. v. Vliet, and M. Shahin, "How do open source communities document software architecture: An exploratory survey," in *19th International Conference on Engineering of Complex Computer Systems*, 2014, pp. 136–145.
- [7] B. Nuseibeh, S. Easterbrook, and A. Russo, "Leveraging inconsistency in software development," *Computer*, vol. 33, no. 4, pp. 24–29, 2000.
- [8] R. Wohlrab, U. Eliasson, P. Pelliccione, and R. Haldal, "Improving the consistency and usefulness of architecture descriptions: Guidelines for architects," in *2019 IEEE ICSA*, 2019, pp. 151–160.
- [9] B. Nuseibeh, S. Easterbrook, and A. Russo, "Making inconsistency respectable in software development," *Journal of Systems and Software*, vol. 58, no. 2, pp. 171–180, 2001.
- [10] J. Keim and A. Koziolok, "Towards consistency checking between software architecture and informal documentation," in *2019 IEEE International Conference on Software Architecture Companion (ICSAC-C)*, 2019, pp. 250–253.
- [11] J. Keim, S. Schulz, D. Fuchß, C. Kocher, J. Speit, and A. Koziolok, "Tracelink recovery for software architecture documentation," in *Software Architecture*, S. Biffl, E. Navarro, W. Löwe, M. Sirjani, R. Mirandola, and D. Weyns, Eds. Springer International Publishing, 2021, pp. 101–116.
- [12] P. H. Matthews *et al.*, *Syntax*. Cambridge University Press, 1981.
- [13] J. Keim, S. Schulz, D. Fuchß, and A. Koziolok, "Replication Package for "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery"," 2023. [Online]. Available: <https://doi.org/10.5281/zenodo.7555195>
- [14] T. Olsson and J. Grundy, "Supporting traceability and inconsistency management between software artifacts," *6th International Conference on Software Engineering and Applications*, pp. 484–489, 2002.
- [15] N. Borovits, I. Kumara, D. Di Nucci, P. Krishnan, S. D. Palma, F. Palomba, D. A. Tamburri, and W.-J. v. d. Heuvel, "Findici: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code," *Empirical Software Engineering*, vol. 27, no. 7, p. 178, Sep 2022.
- [16] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, pp. 427–435, May 2021.
- [17] Z. Dong, A. Andrzejak, D. Lo, and D. Costa, "Orplocator: Identifying read points of configuration options via static analysis," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (IS-SRE)*, 2016, pp. 185–195.
- [18] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. Gall, "Automatic detection and repair recommendation of directive defects in java api documentation," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 1004–1023, 2020.
- [19] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.
- [20] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards detecting inconsistent comments in java source code automatically," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020, pp. 65–69.
- [21] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Emp. Softw. Engg.*, vol. 21, no. 2, p. 565–604, 2016.
- [22] A. Egyed, "Scalable consistency checking between diagrams - the viewintegra approach," in *16th Annual International Conference on Automated Software Engineering (ASE 2001)*, 2001, pp. 387–390.
- [23] A. Kozlenkov and A. Zisman, "Are their design specifications consistent with our requirements?" in *Proceedings IEEE Joint International Conference on Requirements Engineering*, 2002, pp. 145–154.
- [24] V. Gervasi and D. Zowghi, "Reasoning about inconsistencies in natural language requirements," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 3, p. 277–330, jul 2005.
- [25] A. Fantechi and E. Spinicci, "A content analysis technique for inconsistency detection in software requirements documents." 2005, pp. 245–256.
- [26] M. Kamalrudin, J. Grundy, and J. Hosking, "Managing consistency between textual requirements, abstract interactions and essential use cases," in *IEEE Annual Computer Software and Applications Conference*, 2010, pp. 327–336.
- [27] R. Ali, F. Dalpiaz, and P. Giorgini, "Reasoning with contextual requirements: Detecting inconsistency and conflicts," *Information and Software Technology*, vol. 55, no. 1, pp. 35–57, 2013.
- [28] I. Lytra and U. Zdun, "Inconsistency management between architectural decisions and designs using constraints and model fixes," in *2014 23rd Australian Software Engineering Conference*, 2014, pp. 230–239.
- [29] S. Çiraci, H. Sözer, and B. Tekinerdogan, "An approach for detecting inconsistencies between behavioral models of the software architecture and the code," in *IEEE 36th Annual Computer Software and Applications Conference*, 2012, pp. 257–266.
- [30] J. J. Li and J. R. Horgan, "To maintain a reliable software specification," in *International Symposium on Software Reliability Engineering*. IEEE, 1998, pp. 59–68.
- [31] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8. Soviet Union, 1966, pp. 707–710.
- [32] D. Charlet and G. Damnati, "SimBow at SemEval-2017 task 3: Softcosine semantic similarity between questions for community question answering," in *11th International Workshop on Semantic Evaluation (SemEval-2017)*. ACL, 2017, pp. 315–319.
- [33] W. E. Winkler, "String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage." 1990.
- [34] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 728–738, 1984.
- [35] P. Kruchten, "An ontology of architectural design decisions in software-intensive systems," *2nd Groningen Workshop on Software Variability*, pp. 54–61, 2004.
- [36] J. Keim, T. Hey, B. Sauer, and A. Koziolok, "A taxonomy for design decisions in software architecture documentation," Tech. Rep., 2022, international workshop on Mining Software Repositories for Software Architecture at ECSA 2022.
- [37] D. Fuchß, S. Corallo, J. Keim, J. Speit, and A. Koziolok, "Establishing a benchmark dataset for traceability link recovery between software architecture documentation and models," in *2nd International Workshop on Mining Software Repositories for Software Architecture - Co-located with 16th European Conference on Software Architecture*, 2022.
- [38] T. Olsson, M. Ericsson, and A. Wingkvist, "Semi-automatic mapping of source code using naive bayes," in *13th European Conference on Software Architecture*. Association for Computing Machinery, 2019, pp. 209–216.
- [39] T. de Jong and J. M. E. M. van der Werf, "Process-mining based dynamic software architecture reconstruction," in *13th European Conference on Software Architecture*. Association for Computing Machinery, 2019, pp. 217–224.
- [40] F. G. Toosi, J. Buckley, and A. R. Sai, "Source-code divergence diagnosis using constraints and cryptography," in *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. Association for Computing Machinery, 2019, pp. 205–208.
- [41] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, p. 4, 2006.
- [42] J. Cleland-Huang, O. Gotel, A. Zisman *et al.*, *Software and systems traceability*. Springer, 2012, vol. 2, no. 3.
- [43] H. Cramér, "Mathematical methods of statistics." 1946.
- [44] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

- [45] D. Chicco, "Ten quick tips for machine learning in computational biology," *BioData mining*, vol. 10, no. 1, pp. 1–17, 2017.
- [46] D. Chicco, N. Tötsch, and G. Jurman, "The matthews correlation coefficient (mcc) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation," *BioData mining*, vol. 14, no. 1, pp. 1–22, 2021.
- [47] G. A. Ferguson, "The factorial interpretation of test difficulty," *Psychometrika*, vol. 6, no. 5, pp. 323–329, 1941.
- [48] J. Ernest C. Davenport and N. A. El-Sanhurry, "Phi/phimax: Review and synthesis," *Educational and Psychological Measurement*, vol. 51, no. 4, pp. 821–828, 1991.
- [49] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy, "Improving traceability link recovery using fine-grained requirements-to-code relations," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 12–22.
- [50] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," vol. 14, no. 2, p. 131.
- [51] X. Wang, Y. Jiang, N. Bach, T. Wang, Z. Huang, F. Huang, and K. Tu, "Automated concatenation of embeddings for structured prediction," *arXiv preprint arXiv:2010.05006*, 2021.
- [52] I. Yamada, A. Asai, H. Shindo, H. Takeda, and Y. Matsumoto, "LUKE: Deep contextualized entity representations with entity-aware self-attention," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Nov. 2020, pp. 6442–6454.
- [53] X. Wang, Y. Jiang, N. Bach, T. Wang, Z. Huang, F. Huang, and K. Tu, "Improving named entity recognition by external context retrieving and cooperative learning," *arXiv preprint arXiv:2105.03654*, 2021.