Recovering Trace Links Between Software Documentation And Code

Jan Keim
jan.keim@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Sophie Corallo
sophie.corallo@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Dominik Fuchß
dominik.fuchss@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Tobias Hey
hey@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Tobias Telge
tobias.telge@alumni.kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Anne Koziolek
koziolek@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

ABSTRACT

Introduction Software development involves creating various artifacts at different levels of abstraction and establishing relationships between them is essential. Traceability link recovery (TLR) automates this process, enhancing software quality by aiding tasks like maintenance and evolution. However, automating TLR is challenging due to semantic gaps resulting from different levels of abstraction. While automated TLR approaches exist for requirements and code, architecture documentation lacks tailored solutions, hindering the preservation of architecture knowledge and design decisions.
Methods This paper presents our approach TransArC for TLR between architecture documentation and code, using component-based architecture models as intermediate artifacts to bridge the semantic gap. We create transitive trace links by combining the existing approach ArDoCo for linking architecture documentation to models with our novel approach ArCoTL for linking architecture models to code.
Results We evaluate our approaches with five open-source projects, comparing our results to baseline approaches. The model-to-code TLR approach achieves an average F_1-score of 0.98, while the documentation-to-code TLR approach achieves a promising average F_1-score of 0.82, significantly outperforming baselines.
Conclusion Combining two specialized approaches with an intermediate artifact shows promise for bridging the semantic gap. In future research, we will explore further possibilities for such transitive approaches.

CCS CONCEPTS

• **Software and its engineering** → *Software design engineering*; **Documentation**; *Maintaining software*; *Software evolution*; **Software architectures**; • **Computing methodologies** → **Natural language processing**; **Information extraction**.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639130

KEYWORDS

software traceability, software architecture, documentation, transitive links, intermediate artifacts, information retrieval

ACM Reference Format:
Jan Keim, Sophie Corallo, Dominik Fuchß, Tobias Hey, Tobias Telge, and Anne Koziolek. 2024. Recovering Trace Links Between Software Documentation And Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639130

1 INTRODUCTION

During software development, various artifacts are created. These artifacts are at different levels of abstraction and cover (partially) different aspects of a system. The problem is that the relationships between the artifacts are not always apparent and, thus, cannot be used. Making these relationships explicit improves software quality and simplifies processes. As such, software traceability is an important factor in successful software development. With traceability link recovery (TLR), software engineers can connect any uniquely identifiable software engineering artifacts by creating explicit trace links, maintain these trace links, and use the resulting network to gain knowledge about the software product and its development [8].

Therefore, software quality can be improved by creating and maintaining trace links [55]. For example, in collaborative development, TLR can help engineers to keep all artifacts synchronized and consistent [51]. Furthermore, traceability supports numerous critical software engineering tasks (cf. [7]). For example, trace links can improve the efficiency of software maintenance [33, 39], bug localization [53], change impact analysis [10], and system security [44, 46]. Trace links are also used to demonstrate the safety of systems [38, 40, 45]. Some standards, such as ISO 26262 about the functional safety of road vehicles, even mandate traceability.

Despite all the benefits, the main drawback of traceability is the time-consuming and error-prone process of manually creating and maintaining trace links [9, 52]. This is mainly due to the semantic gap between artifacts of different abstraction levels [3], e.g., requirements and code. Some (semi-)automated approaches have been designed to assist users but face the same challenge.

Likewise, automated TLR approaches often only look at requirements and code. However, other types of artifacts, such as the documentation of the system's software architecture, are just as relevant. A software's architecture is key to successfully developing,

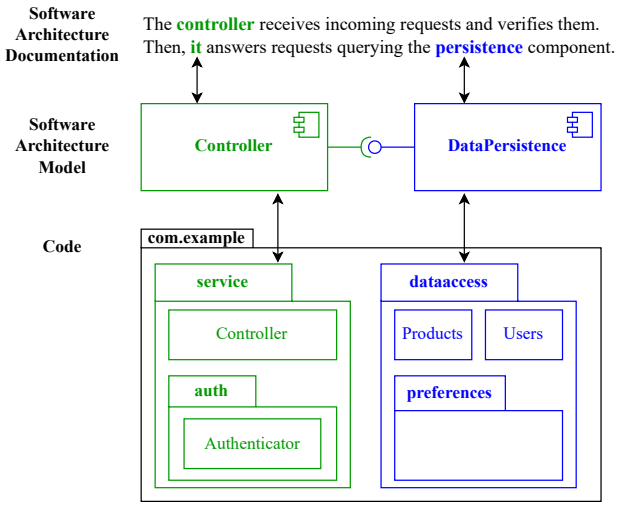

Figure 1: Running example: System with software architecture documentation, software architecture model, and code[1].

maintaining, and evolving the system [41]. Software architecture documentation (SAD) preserves the knowledge about the architecture, such as the underlying design decisions. Documentation prevents rapid deterioration [50, 65] and enhances the benefits of a good software architecture. Using trace links to find relevant knowledge in SADs further enhances the utility of software architecture while improving the usability for developers and architects. Therefore, linking SADs to other software artifacts, such as other design artifacts or code, is beneficial on several levels.

Because different types of artifacts cover different aspects and have different views on the system, these artifacts have different levels of abstraction, and, as such, there is a semantic gap between them [3]. Automated approaches have been developed to bridge this gap by capturing the underlying semantics. However, this task is inherently challenging, and such approaches are prone to misinterpretation, leading to potential imprecision. To overcome the semantic gap between source and target artifacts, some other approaches suggest using intermediate artifacts (cf. [2, 44, 47, 64]). The idea is that those intermediate artifacts have a smaller semantic gap to the artifacts, allowing for easier pairing. For example, design documentation is semantically closer to design artifacts like software architecture models (SAMs) than to code. At the same time, SAMs are also closer to code and serve as an intermediate artifact between SADs and SAMs. For those smaller semantic gaps, more specialized approaches show promising results (cf. [20, 25, 59]). Thus, our main idea is to chain trace links from multiple specialized approaches to recover trace links for intermediate artifacts and transitively combine the resulting links. For example, we can use one approach to recover links between SADs and SAMs, and another one to recover trace links between SAMs and code. We can then transitively combine the links to obtain trace links between SADs and code, bridging the semantic gap.

Figure 1 depicts an example of SAD to SAM to code[1]. We observe two key components: the *Controller* on the left and the *DataPersistence* on the right. The SAD details their respective responsibilities and their structural relationship is visualized in the SAM. The code implementation introduces two essential packages, namely `service` and `dataaccess`, each of which includes sub-packages such as `auth` and `preferences`. The `Controller` class effectively employs the `Authenticator` to verify incoming requests. The classes `Products` and `Users` serve as repositories, leveraging information from hidden classes within the `preferences` package to establish connections with a database.

In the example, elements that should be linked with trace links are marked with the same color. Consequently, both sentences, due to the references to the terms "controller" and "it", are associated with the *Controller* component and the classes within the `service` package. Similarly, the second sentence is linked to the *DataPersistence* component and the classes in the `dataaccess` package.

The example highlights several challenges encountered during the creation of trace links. For instance, the description of the *controller* component is necessary to properly link it to the whole `service` package instead of only the `Controller` class. Another challenge is the changed naming of the "persistence" component in the SAD to `dataaccess` in the code. In such cases, using the *DataPersistence* component in the SAM can help to bridge the gap.

This paper presents our novel approach using transitive links to recover trace links between SADs and code. To bridge the semantic gap between documentation and code, we use intermediate artifacts in the form of component-based SAMs that describe the structure of the system, such as UML component diagrams. The idea is to combine two specialized approaches to improve the results for the wider semantic gap between the original artifacts. More specifically, we concatenate an approach for linking SADs and SAMs with an approach for linking SAMs to code. To link SADs and SAMs, we use the existing Architecture Documentation Consistency (ArDoCo) approach by Keim et al. [25] as it is the state of the art for this problem. For linking SAMs and code, we present a new approach called *ARchitecture-to-COde Trace Linking (ArCoTL)*. The ArCoTL approach uses several heuristics and aggregation methods that are concatenated in a graph to recover trace links. To be independent of input languages and to support different component-based architecture description languages and programming languages, the approach uses abstractions for code and architecture. The approach that transitively combines ArDoCo and ArCoTL to recover trace links between SADs and code is called *Transitive links for Architecture and Code (TransArC)* (see Figure 2).

A drawback of our approach is the need for an additional type of artifact, the component-based SAMs. We argue that SAMs as artifacts serve additional purposes besides our TLR approach. Surveys with practitioners also indicate that a considerable fraction of practitioners already have SAMs: 86% of practitioners used UML as architecture description language in a study by Malavolta et al. [35] and 26.3% of practitioners used architecture modeling or visualization with UML in a study by Tian et al. [63]. In addition, there are approaches that (semi-) automatically recover SAMs from code

[1]For readability reasons, we use an abstract representation of the code in this example instead of the actual source code our tool uses.

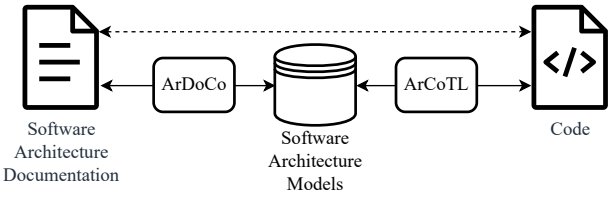

Figure 2: High-level view of the TransArC approach for linking SADs and code using ArDoCo [25] and our novel ArCoTL.

and deployment artifacts (cf. [5, 11, 27, 30, 61]), reducing the overhead to create SAMs. In summary, the required SAMs potentially improve results because they reduce the semantic gap, can likely be created in a lightweight manner, and enable additional tasks.

Overall, this paper presents our transitive approach to linking architecture documentation and code, including an approach for linking models and code. Consequently, we have the following research questions:

- **RQ1** How well can our approach ArCoTL recover trace links between component-based software architecture models and code?
- **RQ2** How accurate can our approach TransArC using intermediate artifacts recover trace links between software architecture documentation and code?
- **RQ3** How do the results for linking software architecture documentation and code compare to state-of-the-art requirements-to-code approaches?

The main contributions of this paper are:

- **C1** We present an approach for linking component-based software architecture models and code.
- **C2** We combine two specialized TLR approaches to transitively link software architecture documentation and code using architecture models as intermediate artifacts.
- **C3** We provide code, baselines, evaluation data, and results in a replication package [26].

The remainder of the paper is structured as follows: Related work is examined in Section 2. In Section 3, we provide basic information about the approach ArDoCo by Keim et al. [25] that we use for linking SADs and SAMs. Our detailed approach is presented in Section 4. We evaluate our approach in Section 5 and discuss threats to validity in Section 6. Lastly, we conclude this paper in Section 7.

2 RELATED WORK

Automated software and system traceability encompasses different domains, use cases, and techniques [2, 62]. Most of the successful techniques are based on *information retrieval (IR)* and *machine learning (ML)* that we look into in Section 2.1 and Section 2.2, respectively. As we employ *transitive tracing* techniques, we also look into it in Section 2.3. Overall, we focus on related work that involves similar source and target models to texts and source code. In these sections, we also provide details for the approaches by Gao et al. [15] and by Hey et al. [23] as well as CodeBERT [12] that we use as baselines in our evaluation (cf. Section 5.3).

2.1 Information Retrieval-based TLR

Early breakthroughs of TLR were primarily based on IR techniques. Initial approaches utilized general preprocessing and combinations of probabilistic models and vector space models (VSM) to identify candidates for trace links. An example of such an approach is presented by Antoniol et al. [1], who create trace links between code and source code documentation. However, a common challenge in this context is handling semantically similar expressions like synonyms. Various methods have been proposed to address this problem, including the use of word embeddings [6, 23, 67], latent semantic indexing [36], incorporation of synonym coefficients for similarity calculations [19], and the construction of semantic-relationship graphs [60], among others.

VSM remains a popular technique for TLR [16, 34]. However, since VSM are primarily designed for text-based data, several approaches try to reuse structural information to capture contexts in source code [29, 49]. These approaches have shown promise in improving TLR between requirements and source code [29].

Lohar et al. [32] compare various combinations of IR techniques to create trace links between different artifact types, including requirements and code. In their evaluation covering different projects and domains, their approach achieved high mean average precisions (MAPs) of 0.80 and 0.86 for recovering trace links between use or test cases and code. However, the authors observed a strong dependency between configurations, projects, and artifacts, as the best results for projects were attained with different configurations.

To address these challenges, the approach TAROT introduces the use of so-called biterms to improve TLR between requirements and code [15]. Biterms, in the textual context, refer to two terms within a sentence that have a grammatical relationship. Similarly, biterms in the code side represent any combinations of two terms within identifiers, with code comments treated analogously to text. The intersection of both biterm sets results in consensual biterms, which are subsequently weighted based on their frequency and location. Different IR models, including VSM, latent semantic indexing (LSI), and probabilistic Jensen Shannon model divergence (JSD), are employed to create candidate trace links between requirements and code. In their evaluation, TAROT achieves a MAP of 0.62 for the iTrust project when using VSM. When TAROT is combined with the CLUSTER enhancing strategy (cf. [14]), the MAP is improved to 0.73 using LSI.

Hey et al. [23] leverage IR-based metrics to retrieve trace links between requirements and source code classes. However, their approach FTLR stands out from other works as it employs a more fine-grained technique. In FTLR, both artifact types are split into smaller units, with requirements divided into sentences and code elements represented by their public method signatures, extended by the name of the containing class and their documenting comments.

The elements of both artifact types are preprocessed. The preprocessing steps include stop word removal, lemmatization, and word length filtering. The resulting artifact element types are represented as Bag-of-Embeddings, using the pre-trained word embedding representations of fastText [42].

Finally, trace links are generated in two steps. First, the fine-grained elements are mapped based on the Word Mover's Distance (WMD) of the Bag-of-Embeddings. Second, the resulting element

trace links are filtered using a threshold. FTLR then creates trace links for a class based on a majority vote, linking it to the most frequently mapped requirements among its methods. With the best configuration, the authors report an average F_1-score of 0.327 for this approach in their evaluation.

Overall, IR-based TLR approaches can achieve accurate results when the semantic gap is small, but can struggle with larger ones. In this study, we explore how intermediate artifacts can help to bridge such larger semantic gaps.

2.2 Machine learning-based TLR

As for most areas, the progress made by ML and language models has significantly advanced TLR approaches. ML is applied in various ways, including combining recurrent neural networks (RNNs) and word embeddings [17], combining feedforward neural networks and cluster-pair rank models [66], ranking embeddings [68], using pre-trained language models, such as CodeBERT [31], and using pre-trained classifiers based on textual features [54]. Other researchers employ active learning [43] to tackle the challenge of the availability of training data. However, these approaches use initial trace links of the projects to train their models and therefore tackle a different kind of TLR problem than our approach. Their out-of-the-box performance for recovering trace links of unseen projects without initial trace links is unclear. As large language models can be used for transfer learning across tasks and projects, they are promising candidates for the TLR task tackled in this paper. Therefore, we use CodeBERT [12] as a baseline besides the two previously mentioned IR-based techniques.

2.3 Utilizing intermediate artifact types

Instead of attempting to bridge large semantic gaps, some approaches suggest using transitive trace links. The Connecting Links Method (CLM) focuses on establishing transitive trace links between two artifacts using a third artifact [47, 64]. In this process, CLM generates trace links to the intermediate artifact by employing a VSM. When two trace links share the same element in the intermediate artifact, they are connected.

A similar strategy is adopted by COMET, which leverages a hierarchical probabilistic model to infer trace links [44]. The process involves several steps: (1) initial trace links are generated using IR and ML techniques; (2) these links are reviewed by developers in a second stage; (3) the final phase operates on an approximation of the previous steps. Transitive links between two artifacts are established when two links from one artifact refer to the same element in another artifact. The similarity for the intermediate element is derived either from textual similarity (for requirements) or from execution traces (for test cases).

In a different approach, Rodriguez et al. [57] utilize existing artifacts as intermediates. For instance, design artifacts are used to map requirements to subsystem requirements. Unlike COMET, the transitive trace links in this approach can connect different artifacts. In their evaluation, Rodriguez et al. compare LSI and VSM as techniques for direct trace links with transitive trace links. The transitive trace links are retrieved by applying different methods to generate the required trace links to the intermediate artifacts. Additionally, they compare these results to hybrid approaches, which combine the results of direct and transitive trace links. The evaluation demonstrates that, in most cases, the best transitive techniques significantly outperform the direct techniques ($\alpha \leq 0.001$). Although hybrid approaches often slightly outperform transitive approaches, this difference was not statistically significant. This shows that transitive approaches for TLR are helpful for improving the performance, supporting our idea.

As such, we use some ideas from these approaches and apply them to our TLR scenario. Moreover, we adapt them using different approaches for the specific intermediate steps to benefit from specialized approaches.

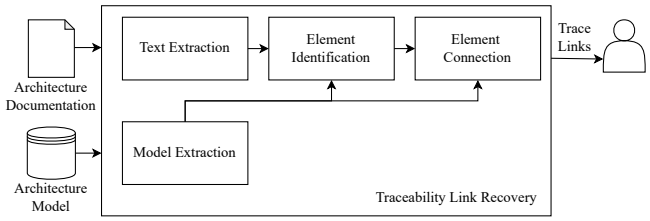

Figure 3: Overview of the ArDoCo approach [25]

3 BACKGROUND: ARDOCO

As this work uses ArDoCo [25], we introduce their approach in this section. ArDoCo is a tool for TLR and inconsistency detection between natural language SADs and SAMs like UML.

The tool is designed as an extendable pipeline that incorporates agent-based heuristics. The pipeline, depicted in Figure 3, comprises various steps for TLR. The first two pipeline steps process the artifacts individually and independently.

Initially, the approach processes the input SAD text using traditional natural language processing (NLP) techniques, including part-of-speech tagging, sentence splitting, lemmatization, and dependency parsing. These techniques aid in extracting structural information from the text.

Next, the *text extraction* step identifies name- and type-like mentions from the processed text. For instance, in the running example in Figure 1, words like "controller", "persistence", or "component" are extracted. Similar mentions, i.e., identical or very similar words are clustered, similar to coreference resolution. Each cluster is annotated with confidence values that indicate whether the underlying element is considered a *name* or a *type*. To achieve this, ArDoCo leverages phrase structures (cf. [37]). In the running example, the approach would confidently annotate "component" as a type, while the other two highlighted words are likely names.

The *model extraction* step analyzes the SAM and its meta-model, extracting elements like types and architecture components.

Following this, the *element identification* identifies potential model elements based on the mentions in the text and the extracted information from the meta-model. This process creates so-called recommended instances, which can be seen as special named entities representing potential model elements.

Finally, the *element connection* establishes trace links between the SAD and the SAM by mapping recommended instances to similar elements of the instantiated model. To calculate similarity, ArDoCo

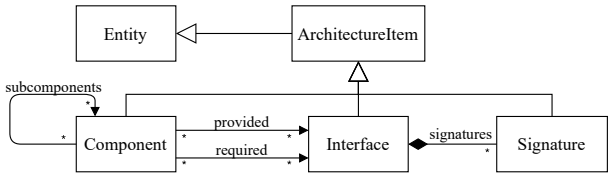

Figure 4: Intermediate model for architecture models

provides different metrics, including string-based, vector-based, and other approaches.

For their evaluation, Keim et al. use five open-source projects to assess the performance of their approach. The overall evaluation results show an average precision of 0.87, an average recall of 0.82, and an average F_1-score of 0.82.

4 APPROACH

Our approach TransArC to recover trace links between SAD and code involves three steps, incorporating two distinct approaches (see Figure 2). In the initial step, we utilize the ArDoCo approach, as presented by Keim et al. [25] (see Section 3), to establish links between SAD and SAM. Moving on to the second step, we introduce our novel ArCoTL approach for linking SAM to code. The approach transforms the artifacts into generalized intermediate representations and computes trace links using a computational graph. Detailed insights into this approach are provided in Section 4.1. In the final step, we combine the outcomes of both approaches in a transitive manner to create the desired trace links between SAD and code. The procedure for this step is elaborated in Section 4.2.

4.1 ArCoTL: Linking SAM to Code

Our strategy for establishing links between SAM and code involves a two-step approach, namely ArCoTL.

In the first step, we transform both types of artifacts into generalized intermediate representations. In the second step, we employ various heuristics and aggregations to examine potential hints that signify a link between an architecture element and a code element. To combine the different heuristics, we use a computational graph.

4.1.1 Intermediate Representations. The primary objective of intermediate representations for both artifacts is to provide an abstraction from the concrete languages used for architecture description and programming: First, they capture the essential commonalities present in each respective part, enabling us to simplify complex inputs and focus solely on the aspects that are required for further processing. Second, these representations allow us to define heuristics at a more abstract level, free from the intricacies of individual languages. As a result, our heuristics become independent of the particular input languages used, offering greater flexibility and applicability.

Our representation for architecture models, as depicted in Figure 4, is characterized by its simplicity. The architecture model consists of architecture items subdivided into components, interfaces, and signatures. Components may contain sub-components and can provide or require interfaces that have signatures.

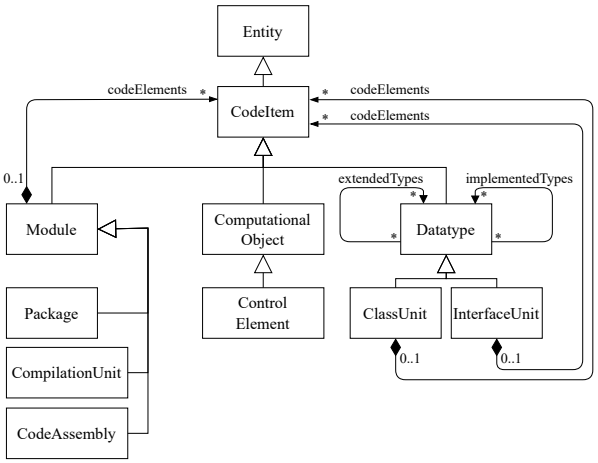

Figure 5: Intermediate model for code

The code model in Figure 5 is similarly straightforward. We base the model on the Knowledge Discovery Metamodel (KDM) by the OMG [48] and are using an excerpt of the KDM for our code model.

Code items fall into three categories: modules, computational objects, and datatypes. Modules encompass packages, compilation units, or code assemblies, representing the structural elements of code. Computational objects represent functional parts of the code, such as methods. Lastly, the code model includes datatypes, specifically classes and interfaces that can exist within the code.

We designed our models to maintain simplicity and generality, enabling easy integration of additional architecture description languages or programming languages through adapters. These adapters transform the input into our intermediate representations, representing either the architecture or code model. The advantage of this approach is that an adapter only has to be defined once for each language, eliminating the need for adaptations in other parts of our methodology, such as the computational graph with its associated heuristics. Currently, our implementation supports UML components diagrams and PCM [56] as architecture description languages, along with Java and Bash as programming languages. The rationale behind selecting these languages is to demonstrate our approach's versatility and applicability with different languages.

Leveraging these intermediate representations, we are also able to define trace links formally: A trace link comprises an architecture item paired with a corresponding code item. To facilitate our calculations, we construct a repository representing the traceability matrix, encompassing all combinations of architecture and code items. This matrix is the foundation for our traceability analysis.

4.1.2 Computational Graph. Our approach employs a variety of heuristics and aggregators for conducting its calculations. The heuristics evaluate each pair of entities, i.e., architecture and code items. We categorize our heuristics into two distinct types: standalone and dependent heuristics. The outputs of each heuristic are *mappings* that contain each entity pair together with the heuristic's confidence for this pair. The aggregators combine the mappings

and filter out unlikely pairs. We organize the calculations in a computational graph. The outputs of the computational graph are the identified trace links, i.e., pairs of architecture and code entities.

Algorithm 1 StandaloneHeuristic$_H$(archItems, codeItems)

```
mappings ← ∅
for all archItem ← archItems, codeItem ← codeItems do
    confidence ← similarity_H(archItem,codeItem)
    mappings ← mappings ∪ (archItem, codeItem, confidence)
return mappings
```

Standalone heuristics operate independently, not relying on other heuristics for their execution. Thus, they get the input artifacts, i.e., pairs of architecture and code items, and calculate their similarity to create mappings, as is shown in Algorithm 1. We use the following standalone heuristics:

Package	Compares package name with name of components
Path	Compares the path of a compilation unit with the names of components
Method	Compares method names with names of signatures
Names	Compares names of architecture elements with names of compilation units and datatypes

When comparing entity names, we determine if one entity's name contains parts of the other entity's name. We identify parts of an entity based on word boundaries and camel casing, hyphenations, underscores, periods, and similar separators. However, we disregard capitalization during the equality checks. The approach derives confidence in this comparison from the ratio of the contained parts. Consequently, this comparison is asymmetric. For instance, the name "DatabaseAdapter" contains the name "database", resulting in a similarity score of 50%. Conversely, no containment in the opposite direction leads to no similarity.

This asymmetric comparison is vital for controlling which entity can be contained within another and for increasing precision, especially for heuristics like *Package* or *Path*, where containment plays a crucial role.

Many programming languages encode package names in the path structure. However, slight differences can exist, so we employ both heuristics and incorporate this relationship. To avoid redundant consideration of the package name via both the name and the path, we exclude the package from the path. For example, given the package `mediastore.persistence` and the path *ms-database/src/main/java/mediastore/persistence*, we only utilize the beginning of the path, i.e., *ms-database/src/main/java*. In this example, a difference between both heuristics becomes apparent: The beginning of the path represents the folder that can indicate the component name and provide additional information compared to only the package name. This situation can arise if the project utilizes multiple modules.

Dependent heuristics use the resulting mappings of previous heuristics as their input. As shown in Algorithm 2, dependent heuristics check for affected mappings and calculate the updated confidence. There are the following dependent heuristics:

Hint Inheritance	Inherits results from other heuristics (mappings) along extends- and implements-relations.

Algorithm 2 DependentHeuristic$_H$(mappings)

```
for all m ← mappings do
    for all m_a ← affectedMappings_H(m) do
        m_a.confidence ← getUpdatedConfidence_H(m_a, m)
return mappings
```

Common Words	Checks if names differ only in common words or prefixes/suffixes (e.g., Test, Impl, I).
Amb. Sub-pkg.	Detects ambiguously mapped sub-packages.
Component rel.	Looks at relations between components to resolve ambiguity.
Interface prov.	Checks if a *provide*-relation of the architecture exists in the source code.

The *Hint Inheritance* heuristic uses the assumption that there is a strong coupling between a class and its parent. Consequently, this heuristic inherits mappings that regard a class and applies them to the extending or implementing class.

The *Common Words* heuristic adapts the results of the *Names* heuristic. When two entities' names only differ in common words, prefixes, or suffixes, the approach increases the confidence score. For example, common words include "Test", "Exception", and "Factory". We regard well-known prefixes and suffixes, e.g., "Impl", used to indicate implementations of abstract classes.

When dealing with packages, it is possible for a component and its packages to be included in the packages of another component, creating an ambiguous mapping. For instance, the package `dataaccess.preferences` can be mapped to both the *DataPersistence* and the *Preferences* component. The *Ambiguous Sub-package* heuristic is employed to address this ambiguity in two possible cases: In the first case, when the heuristic finds another location where the component (e.g., *Preferences*) is implemented, it is assumed that there is no actual relation between the package (`dataaccess.preferences`) and said component. We reason that the implementation of a component should not be scattered in the code. Consequently, the heuristic revokes the mapping to the (*Preferences*) component. In the second case, if the approach finds no other location, the sub-package likely is a component (here: *Preferences*), and the heuristic removes the mapping to the *DataPersistence* component.

The *Component relation* and the *Interface provision* heuristics are used to check for relations between components and/or interfaces that can be detected in the code. These heuristics adapt previous results based on the existence and absence of such relations. They play a significant role if multiple mappings exist between an architecture element and code elements. In these cases, they can help remove mappings with source code elements that cannot be associated with the relations in the architecture.

Aggregators. To combine mappings, we adopt two types of aggregators: combiners and selectors. The first category, combiners, consists of aggregators that, as the name suggests, combine the mappings generated by one or more heuristics about the same entity pair. The *Max* combiner utilizes a maximum function, which sets the confidence score of an entity pair to the highest confidence of mappings for this pair. We use the maximum function to indicate that one result of competing heuristics covering different

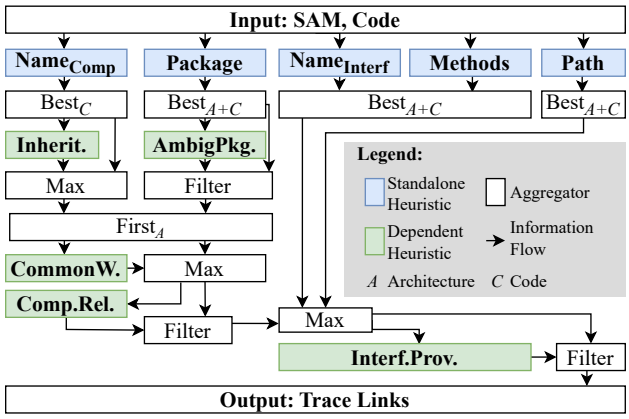

Figure 6: The computational graph of ArCoTL

aspects is sufficient and to avoid negative influence from single low confidence mappings.

The second category of aggregators, selectors, facilitates the selection of mappings based on specific criteria. The *Best* criterion focuses on the mappings of multiple heuristics concerning one certain fixed entity and selects only the highest confidence hint. Consequently, we have two variants of this criterion, one focussing on code items and one on architecture items. The *First* criterion operates by examining the order of the heuristics and selecting, for one entity, the first mapping from an ordered list of heuristics. Consequently, the mappings are assessed sequentially for each entity, and a mapping is chosen over others if its confidence is greater than 0. We base this approach on the idea that heuristics and, thus, the mappings are not equally important and need to be weighted. We do this weighting according to their order, which proves particularly relevant for specific combinations.

Lastly, we employ *filters* to remove mappings that contain negative evidence from dependent heuristics such as *Ambiguous Subpackage*, *Component relation*, and *Interface provision*. These filters play a crucial role in refining the final results of our approach.

Computational graph. The computational graph in Figure 6 constitutes the backbone of ArCoTL. It comprises our five standalone heuristics as the entry points into the computation process. Mappings from these heuristics are then filtered with the *Best* criterion.

Initially, our approach selects the best code items for each architecture item, i.e., the set of code items that share the highest confidence for each corresponding architecture item. Subsequently, the computation proceeds with the inverse procedure, selecting the best architecture item for the previously selected code items.

For the *Names* heuristic for components, filtering is based solely on the best architecture items for each code item. We intentionally skip the other direction, as architecture elements are generally implemented with one or more code elements, and limiting architecture items to only the best code items could hinder the recall of the results. In general, we apply both directions to have precise result pairs and only one direction for exploration.

We make use of one *First* matcher when combining the *Package*- and *Names*-parts, prioritizing package names. Here, we first look at the package name to leverage our underlying assumption that package structures and component structures might exhibit similarities. Additionally, some names can also be part of the package. Therefore, we first select the results of the *Package* heuristic and resort to the *Names* heuristic otherwise.

We use various dependent heuristics to adjust confidence values and filter out improbable mappings as part of the calculation process. Toward the end, the approach merges the computation results using the maximum combiner to select only mappings with the highest confidence for each entity pair. The results are further filtered using the *Interface provision* heuristic. This refinement step enhances the precision of the final results.

Finally, we generate trace links for each remaining mapping, i.e., pair of architecture item and code item with confidence > 0.

4.2 TransArC: Transitively Link SAD and Code

To finally establish trace links between documentation and code, we combine two distinct approaches for our approach TransArC: ArDoCo [25], which facilitates trace links between SADs and SAMs, and our approach ArCoTL for linking SAMs to code.

The combination of these two approaches is achieved by creating transitive links. This process involves linking SADs to code by aggregating the trace links from the other two approaches that share the same model elements from the SAM. This integration of specialized approaches allows us to bridge the semantic gap more effectively, particularly when linking SAMs and code. We can improve the results by leveraging the structural information in SAMs, which may not be as explicitly represented in SADs.

While transitive links provide comprehensive traceability, there might be instances where code entities are mentioned in the documentation but cannot be directly linked to the design artifact. We could utilize existing IR approaches to establish the missing links in such cases. However, based on our experience, architecture documentation seldom refers directly to code entities (e.g., classes), as they are typically not represented in design artifacts. Architecture documentation typically emphasizes architecture components over implementation details. To avoid introducing imprecision, we currently refrain from applying such approaches.

5 EVALUATION

This section evaluates our approaches for TLR between code, SAMs, and documentation to answer our research questions. Overall, we want to demonstrate the performance of our approach ArCoTL for TLR between SAM and code and the performance of our approach TransArC for TLR between SAD and code, including a comparison to state-of-the-art approaches.

We first provide a detailed description of the evaluation dataset in Section 5.1. We introduce and explain the metrics to quantify the performance of our approaches employed in the evaluation in Section 5.2. In Section 5.3, we describe the baseline approaches utilized for comparative analyses. Finally, we present the detailed results of our evaluation in Section 5.4, including the comparative assessment of our approach to the baselines. To benefit the research community and ensure our findings' reproducibility, we make our approach, the baselines, all our results, and the experimental data available in a dedicated replication package [26].

Table 1: Number of artifacts per artifact type and number of trace links in the gold standard for each project

Artifact Type		MS	TS	TM	BBB	JR
SAD	# Sentences	37	43	198	85	13
SAM	# Model elements	23	19	16	24	6
Code	# Files	97	205	832	547	1,979
SAM-Code	# Trace links	60	164	1,616	730	1,956
SAD-Code	# Trace links	50	707	7,610	1,295	8,240

5.1 Dataset

To assess the effectiveness of our approaches, we utilized the benchmark dataset provided by Fuchß et al. [13]. This dataset comprises five open-source projects, each differing in size and domain. The projects are MediaStore (MS), TeaStore (TS), TEAMMATES (TM), BigBlueButton (BBB), and JabRef (JR). The benchmark contains the documentation of the projects created by the respective developers. Additionally, the benchmark contains the development view (cf. [28]) in the form of structural architecture models that originate from other researchers (MS, TS, TM) or are reverse-engineered (BBB, JR) (cf. [13, 25]). The benchmark contains gold standards for the trace links between these projects' SADs and SAMs. These gold standards were created in small user studies.

Since the benchmark projects are open-source, we have access to their source code. Through a rigorous process, we established corresponding gold standards for both scenarios, SAD to code and SAM to code. At least two researchers independently generated the gold standards. We resolved discrepancies through discussions and merged the resulting gold standards. Table 1 gives an overview of the dataset and gold standards.

Gold Standard for TLR between SAMs and Code. The gold standard for TLR between code and SAMs consists of a mapping between the model elements of the SAMs and their corresponding relative paths to the source code files. We carefully map model elements, such as components, to the best-fitting code elements. Due to the difference in abstraction levels between code and SAMs, one model element may be mapped to multiple code elements. For example, an interface *IDownload* can be mapped to multiple interfaces in the code, such as *IDownload* and *IDownloadCache*. Sometimes, the best mapping is not at the file level but at the folder/package level. For instance, in the running example illustrated in Figure 1, we map the component *DataPersistence* to the package `dataaccess`. In such situations, all contained elements are considered part of the component, and we trace them accordingly.

Gold Standard for TLR between SADs and Code. The gold standard for trace links recovery between SADs and code encompasses a mapping between the sentences of the SADs and the corresponding source code files. Once again, we map the relative path to a file or the contents within a folder of the source code to the corresponding sentence in the SAD. For example, the first sentence in the running example depicted in Figure 1 is mapped to the files `Controller.java` and `Authenticator.java`.

5.2 Metrics

For our evaluation, we use the metrics Precision (P), Recall (R), and F_1-score (F_1), their harmonic mean. These metrics are commonly used in TLR and comparable research areas (cf. [7, 19]).

Generally, we define the true positives (TP), false positives (FP), and false negatives (FN) as follows: TPs are found trace links between one artifact and another artifact that are also contained in the gold standard. FPs are found trace links between one artifact and another artifact that are not contained in the gold standard. FNs are trace links between one artifact and another artifact that are contained in the gold standard but not found by the approach.

In addition to these metrics, we present two distinct average values. First, we provide the overall average across all projects, regardless of size. This average offers valuable insight into the expected performance per project. Second, we offer a weighted average considering the number of expected trace links for each project. This weighting enables more profound insights into the anticipated efficacy of an approach per trace link.

5.3 Baseline Approaches

This section outlines the baseline approaches utilized for comparison in our study. Specifically, we have chosen to include the approaches TAROT [15], FTLR [23], and CodeBERT [12]. We further adapt ArDoCo [25] to create trace links between SAD and code.

TAROT [15] and FTLR [23] both represent recent and state-of-the-art IR-based solutions for linking requirements and code, and CodeBERT [12] is a large language model trained on finding the most semantically related source code for a given natural language description. Therefore, all three demonstrate promising results for similar TLR problems (cf. Section 2). Since these approaches can handle natural language input and code, they are well-suited for addressing our specific TLR problem. Moreover, their replication packages[2] allow us to use and adapt these approaches. Despite their ability to handle natural language texts, they are targeted towards different artifact types (method documentation and requirements) than SADs. As SADs, method documentation, and requirements are on different levels of abstraction, the results of these approaches can be negatively influenced and might perform worse if applied to SADs. Still, these are suitable approaches that can be used for comparison in our scenario. As an additional baseline suited towards SADs, we adapt ArDoCo to directly operate with our code models (cf. Section 4.1.1).

TAROT. To be able to apply TAROT as-is to our scenario, we interpret each sentence of the SAD as a requirement. We use a threshold to filter TAROT's similarity matrix to generate trace links. We evaluate TAROT's different IR methods, i.e., JSD, LSI, and VSM. For our comparison, we optimize the threshold for each project and method based on the resulting F_1-score, thus obtaining the most favorable results from TAROT.

FTLR. To be able to use FTLR as a baseline, we utilize the most recent version of FTLR [21, 22]. We treat FTLR similarly to TAROT and interpret the sentences of the SAD as requirements. FTLR offers different modes of operation. We apply the different modes and select the best-suited mode (WMD as similarity measure, taking

[2] https://github.com/huiAlex/TAROT,
https://github.com/tobhey/finegrained-traceability

Table 2: Results for TLR between models and code

Project	Precision	Recall	F_1-score
MediaStore (MS)	0.98	1.00	0.99
TeaStore (TS)	0.98	0.98	0.98
TEAMMATES (TM)	1.00	1.00	1.00
BigBlueButton (BBB)	0.94	0.96	0.95
JabRef (JR)	1.00	1.00	1.00
Average	0.98	0.99	0.98
weighted Average	0.99	0.99	0.99

method comments into account) for comparison. The generation of trace links in FTLR also relies on thresholds. For our comparison, we use the results obtained using the best possible threshold for each project. In our replication package [26], we additionally provide the results for FTLR's default thresholds and further modes of operation for comprehensive comparison and analysis.

CodeBERT. We fine-tune the CodeBERT language model [12] for the Java code search task of the CodeSearchNet dataset [24] to apply CodeBERT in our scenario. This dataset consists of pairs of Java methods and their corresponding method documentation, and the task requires the language model to predict whether certain method documentation belongs to a method implementation. This task can be interpreted as similar to linking sentences in SADs to their corresponding source code classes. To fine-tune the model, we adapted the code provided in the replication package of TraceBERT (T-BERT)[3], as it includes support for applying and evaluating CodeBERT models to the TLR task. In particular, we make use of the SINGLE architecture with online negative sampling, which performed best on the CodeSearchNet dataset [31]. After training on the CodeSearchNet dataset, we use the fine-tuned model to predict links between SAD and code.

Adapting ArDoCo: ArDoCode. To add a baseline that is more geared towards SADs, we adapt ArDoCo to create trace links between SADs and code, calling it ArDoCode. For this, we interpret the intermediate code model as a kind of SAM, again resolving packages to create trace links to the contained elements.

More details on the baselines can be found in our replication package [26].

5.4 Results

In this section, we present the results of our evaluation.

Traceability Link Recovery between SAMs and Code. First, we address our first research question (cf. Section 1), which focuses on the performance of our approach in recovering trace links between software architecture models and code. We present precision, recall, and F_1-score for each project, along with the average and weighted average over all projects in Table 2.

The results of our approach are exceptional, indicating a near-perfect linkage of artifacts. This is expected as SAMs and code are closely related. The approach can utilize many of the similar structures that can be found in both types of artifacts. Ultimately, the semantic gap is relatively small between these artifacts, which is also one of our intentions for the overall approach to link SAD and code transitively via SAMs.

However, we still observe some naming-related issues, which can impact the results. As such, false positives occur due to the similar naming of different elements. For example, in BBB, classes in the folder *bbb-graphql-middleware/demo/client* are erroneously mapped to the *HTML5 Client* interface of BBB because of the similarity of the folder name. Likewise, the class *DbException* in the MediaStore project is incorrectly mapped to the *IDB* interface instead of the intended *DB* component. Regarding TS and BBB, our approach produces some false negatives. Here, the challenge lies in mapping elements with significantly different naming conventions, especially when dealing with abbreviations. For instance, due to too dissimilar naming, the component *BBB web* of BigBlueButton could not be mapped to the folder *bigbluebutton-web*. Automated expansion of abbreviations might improve this (cf. [4, 18]).

Concluding **RQ1**, our approach achieves an **average F_1-score of 0.98 (weighted 0.99)** for the trace links between SAMs and code. According to the classification scheme of Hayes et al. [19], our TLR approach achieves *excellent* results. Moreover, these results represent a solid foundation for our transitive approach.

Traceability Link Recovery between SADs and Code. In the second part of the evaluation, we focus on the TLR between SADs and code. Thus, we aim to address our second research question, evaluating the effectiveness of our approach in recovering trace links between software architecture documentation and code using transitive trace links. Additionally, we compare the results obtained by our approach with those of the baseline approaches to answer our RQ3.

To measure the quality of the recovery and facilitate comparison with the baselines, we calculate the precision, recall, and F_1-score for each project. The baseline approaches are initially designed for different tasks, so we optimize their thresholds to maximize the F_1-score. We then select the results of the mode of operation with the best average F_1-score.

In our evaluation, we find that TAROT's best mode of operation utilizes IR in combination with JSD. For FTLR, the best mode of operation uses WMD along with method comments (cf. Section 5.3). The results of all approaches are shown in Table 3. We highlight the best results for each metric and project.

The baseline approaches perform worse, achieving an average F_1-score of 0.22 for TAROT, 0.21 for FTLR, 0.28 for CodeBERT, and 0.37 for ArDoCode. TAROT has a weighted average F_1-score of 0.29, FTLR of 0.28, and CodeBERT of 0.36. ArDoCode has a weighted average F_1-score of 0.62 due to its good performance on the larger projects TM and JR. All baselines achieve better recall than precision. Comparing the results with the reported results of FTLR [23] for its original task, linking requirements and code, we observe similar outcomes. This suggests a certain degree of similarity in the underlying problem to the extent that the performance of the approaches in both scenarios is comparable to some extent.

Comparing our approach to the baselines, our transitive approach combining two specialized approaches outperforms the baselines with an average F_1-score of 0.82 (weighted 0.87). For MS and TS, we achieve perfect precision, while for JR, we achieve perfect recall. This result highlights the project dependency, influenced by the project's characteristics. These characteristics include the

[3] https://github.com/jinfenglin/TraceBERT

Table 3: Results for TLR between documentation and code

	MS			TS			TM			BBB			JR			Avg.			w. Avg.		
Approach	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
TAROT	.09	.24	.13	.19	.44	.27	.06	.32	.11	.07	.18	.10	.32	**1.0**	.49	.15	.44	.22	.19	.63	.29
FTLR	.15	.26	.19	.19	.25	.21	.06	.30	.10	.04	.42	.07	.32	.93	.48	.15	.43	.21	.19	.59	.28
CodeBERT	.29	.12	.17	.26	.57	.36	.09	.22	.12	.07	.49	.12	.49	.83	.61	.24	.45	.28	.28	.53	.36
ArDoCode	.05	**.66**	.09	.20	**.74**	.31	.37	**.92**	.53	.07	.57	.13	.66	**1.0**	.80	.27	.78	.37	.47	.92	.62
TransArC	**1.0**	.52	**.68**	**1.0**	.71	**.83**	**.71**	.91	**.80**	**.77**	**.91**	**.84**	**.89**	**1.0**	**.94**	**.87**	**.81**	**.82**	**.81**	**.94**	**.87**

similarity and consistency of the names in the two artifacts. Moreover, projects may use similar terms corresponding to different entities, making it hard to link them correctly.

We use Wilcoxon's signed rank test to calculate the significance of our approach's F_1-scores compared to the baselines. TransArC is significantly outperforming the baseline approaches (at the 0.05 level). This outcome can likely be attributed to the larger semantic gap the baseline approaches try to bridge. Moreover, TAROT, FTLR, and CodeBERT are fine-tuned for similar but different scenarios.

Nevertheless, the combination of our two specialized approaches proves highly effective. Accordingly, this methodology facilitates the practical use of TLR (cf. [19]). Comparing the results of ArDoCode and TransArC, intermediate artifacts seem to play a vital role in bridging the semantic gap. We argue that creating intermediate artifacts is worthwhile, especially given their other applications and the possibility to reverse engineer them (cf. Section 1).

In summary, our transitive approach TransArC performs promisingly with an **average F_1-score of 0.82 (weighted 0.87)**. Thus, we can confidently answer **RQ2**: According to the classification scheme of Hayes et al. [19], our approach *excellently* recovers trace links between SADs and code. Additionally, we can briefly answer **RQ3**: Our approach **significantly outperforms** the baselines in all projects concerning precision, recall, and F_1-score.

6 THREATS TO VALIDITY

In this section, we discuss threats to validity based on the guidelines by Runeson and Höst [58].

To ensure *Construct Validity*, we employ commonly used metrics and select projects that have been previously studied in research. Additionally, we deliberately choose projects with diverse characteristics, such as different domains, sizes, and relative sizes of documentation to lines of code. By doing so, we aim to mitigate potential confounding factors that could hinder us from effectively addressing our research questions.

Regarding *Internal Validity*, there is the threat that we examine TLR in a way that there are other influencing factors that affect our evaluation. Moreover, we might misinterpret the cause of certain results, leading to wrong conclusions. To address internal validity concerns, we follow established practices to minimize threats. Specifically, we define and evaluate trace links on a sentence level and map them to code files, akin to requirements-to-code traceability. We use the same selection of open-source projects as Keim et al. [25]. This methodology helps mitigate the risk of selection bias. However, it is essential to acknowledge that open-source projects can vary significantly in code quality, documentation, and consistency. Consequently, noise and errors in the data can potentially impact the evaluation process, leading to inflated or deflated performance metrics. Lastly, we might have misjudged the influence of the semantic gap. To mitigate this and to determine the influence, we adapted ArDoCo to work directly with the code model.

Our research design is subject to certain threats to *External Validity*. First, we focus solely on (structural) component-based architecture models, potentially limiting the generalizability of our findings w.r.t. other architectural paradigms or views. According to Tian et al. [63], this view is commonly used for architecture. The results still may vary if, e.g., more logical descriptions and views of the architecture are used, or if the abstraction levels are more different. Second, among the evaluated projects, there are academic projects designed to mimic real applications, but they may exhibit certain differences, making clear statements about generalizability challenging. While our projects encompass different domains and sizes, they do not fully represent all possible application variants. Certain types of projects or specific project characteristics may be overrepresented or underrepresented in our dataset, potentially skewing the evaluation results and restricting the applicability of our findings to different projects and real-world scenarios.

To address threats to validity regarding *Reliability*, we utilize benchmark datasets previously employed in published research. However, we still need to create a gold standard for TLR between SAM and code, as well as SAD and code. For this, we adopt a rigorous approach where at least two researchers independently generated the gold standards, to reduce a potentially biased influence by a single person. These gold standards are merged through comparison and discussion to decide any differences. Still, it is important to acknowledge possible bias, such as researchers interpreting artifacts differently, which impacts reliability to some extent.

7 CONCLUSION AND FUTURE WORK

In conclusion, this scientific study investigated the use of traceability link recovery (TLR) for linking software architecture documentation (SAD) to code. The approach achieves this utilizing a transitive method that combines the results from two specialized approaches, namely TLR for SAD to component-based software architecture model (SAM) using the ArDoCo approach by Keim et al. [25] and a novel approach for TLR between SAM and code.

The proposed approach for model-to-code-TLR utilizes heuristics and aggregators, seamlessly integrated within a computational graph. The efficacy of this method was evaluated on a dataset comprising five diverse projects. To answer RQ1 (performance for TLR

between SAM and code), the evaluation results for the SAM to code TLR exhibit outstanding performance. Thanks to using both name similarity and structural information, the approach achieved an average F_1-score of 0.98. Answering RQ2 (performance for TLR between SAD and code) and RQ3 (comparison to baselines), the results for transitive links between SAD and code display exceptional results with an average F_1-score of 0.82 (weighted 0.87), surpassing the baseline approaches, including the LLM-based approach CodeBERT, significantly. These results further validate the effectiveness of our approach to bridge the semantic gap. The results also show that LLMs are not always the best solution.

In summary, this research contributes a promising methodology for documentation-to-code linkage, and the remarkable achieved performance demonstrates its potential applicability in real-world scenarios. These findings open up exciting possibilities for improving traceability linkage. To ensure replicability and transparency, we have made available a comprehensive replication package [26], encompassing the implemented approach, baseline models, evaluation data, and the obtained results. By sharing this package, we aim to facilitate the reproduction of our study and enable fellow researchers to validate and build upon our findings.

In our evaluation, we used projects from different domains with SADs in different styles and slightly different levels of abstraction. Therefore, we believe that our results are to a certain degree generalizable. The results can vary with vastly diverging SADs. Additionally, SAMs that do not encapsulate the structural components or focus on a too different abstraction can cause a degradation of our approach's performance. However, the general idea to bridge the semantic gap by using multiple specialized approaches should hold for other artifacts that are semantically in between the original gap. Still, we need to explore this theory further in future work.

Consequently, our plan includes extending the generalizability of our approach in future work. First, we aim to evaluate the approach with additional projects to assess its capabilities in different settings, domains, and scenarios. Second, we will investigate the adaptability of our transitive approach for other instances of TLR, such as linking requirements to code. This will entail exploring and identifying various reasonable and efficient intermediate artifacts.

To improve upon the promising results, we plan to make various improvements in future work to further refine the approach.

For our transitive approach, we prioritize precision by disregarding direct mentions of classes in SADs that cannot be linked to SAMs. In future research, we intend to explore opportunities to combine our approach with other (IR) approaches, to address cases where documentation mentions code entities not explicitly represented in the design artifact. This will further strengthen the overall efficacy and completeness of our approach. Moreover, we plan to extend the inconsistency detection capabilities of ArDoCo [25] to SAD and code using TransArC for establishing the required links.

By focusing on these avenues of enhancement, we aim to create a more robust and versatile TLR framework that can further TLR and aid various software engineering tasks in an array of contexts.

ACKNOWLEDGMENTS

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF). This work was also supported by funding from the topic Engineering Secure Systems of the HGF and by KASTEL Security Research Labs.

REFERENCES

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering* 28, 10 (Oct. 2002), 970–983. https://doi.org/10.1109/TSE.2002.1041053

[2] Thazin Win Win Aung, Huan Huo, and Yulei Sui. 2020. A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3387904.3389251

[3] T.J. Biggerstaff, B.G. Mitbander, and D. Webster. 1993. The concept assignment problem in program understanding. In *[1993] Proceedings Working Conference on Reverse Engineering*. 27–43. https://doi.org/10.1109/WCRE.1993.287781

[4] Shanqing Cai, Subhashini Venugopalan, Katrin Tomanek, Ajit Narayanan, Meredith Ringel Morris, and Michael P. Brenner. 2022. Context-Aware Abbreviation Expansion Using Large Language Models. arXiv:2205.03767 [cs.CL]

[5] Yuanfang Cai, Hanfei Wang, Sunny Wong, and Linzhang Wang. 2013. Leveraging Design Rules to Improve Software Architecture Recovery. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures* (Vancouver, British Columbia, Canada) *(QoSA '13)*. Association for Computing Machinery, 133–142. https://doi.org/10.1145/2465478.2465480

[6] Lei Chen, Dandan Wang, Junjie Wang, and Qing Wang. 2019. Enhancing Unsupervised Requirements Traceability with Sequential Semantics. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. 23–30. https://doi.org/10.1109/APSEC48747.2019.00013

[7] Jane Cleland-Huang, Orlena Gotel, Andrea Zisman, et al. 2012. *Software and systems traceability*. Vol. 2. Springer.

[8] CoEST. 2023. Center of Excellence for Software & Systems Traceability. https://web.archive.org/web/20230518011309/http://www.coest.org/. Accessed: 2023-05-18.

[9] Alexander Egyed, Florian Graf, and Paul Grünbacher. 2010. Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments. In *2010 18th IEEE International Requirements Engineering Conference*. 221–230. https://doi.org/10.1109/RE.2010.34

[10] Davide Falessi, Justin Roll, Jin L.C. Guo, and Jane Cleland-Huang. 2020. Leveraging Historical Associations between Requirements and Source Code to Identify Impacted Classes. *IEEE Transactions on Software Engineering* 46, 4 (2020), 420–441. https://doi.org/10.1109/TSE.2018.2861735

[11] Jean-Marie Favre. 2005. Foundations of model (Driven)(Reverse) engineering. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[13] Dominik Fuchß, Sophie Corallo, Jan Keim, Janek Speit, and Anne Koziolek. 2023. Establishing a Benchmark Dataset for Traceability Link Recovery between Software Architecture Documentation and Models. In *Software Architecture. ECSA 2022 Tracks and Workshops*, Thais Batista, Tomáš Bureš, Claudia Raibulet, and Henry Muccini (Eds.). Springer International Publishing, Cham, 455–464. https://doi.org/10.1007/978-3-031-36889-9_30

[14] Hui Gao, Hongyu Kuang, Xiaoxing Ma, Hao Hu, Jian Lü, Patrick Mäder, and Alexander Egyed. 2022. Propagating Frugal User Feedback through Closeness of Code Dependencies to Improve IR-based Traceability Recovery. *Empir Software Eng* 27, 2 (Jan. 2022), 41. https://doi.org/10.1007/s10664-021-10091-5

[15] Hui Gao, Hongyu Kuang, Kexin Sun, Xiaoxing Ma, Alexander Egyed, Patrick Mäder, Guoping Rong, Dong Shao, and He Zhang. 2023. Using Consensual Biterms from Text Structures of Requirements and Code to Improve IR-Based Traceability Recovery. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3551349.3556948

[16] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. 2011. On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 133–142. https://doi.org/10.1109/ICSM.2011.6080780

[17] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press,

Piscataway, NJ, USA, 3–14. https://doi.org/10.1109/ICSE.2017.9
[18] Hussein Hasso, Katharina Großer, Iliass Aymaz, Hanna Geppert, and Jan Jürjens. 2022. Abbreviation-Expansion Pair Detection for Glossary Term Extraction. In *Requirements Engineering: Foundation for Software Quality*, Vincenzo Gervasi and Andreas Vogelsang (Eds.). Springer International Publishing, Cham, 63–78.
[19] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram. 2006. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE TSE* 32, 1 (Jan. 2006), 4–19. https://doi.org/10.1109/TSE.2006.3
[20] Jane Huffman Hayes, Alex Dekhtyar, Senthil Karthikeyan Sundaram, E Ashlee Holbrook, Sravanthi Vadlamudi, and Alain April. 2007. REquirements TRacing On target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering* 3 (2007), 193–202.
[21] Tobias Hey. 2023. Automatische Wiederherstellung von Nachverfolgbarkeit zwischen Anforderungen und Quelltext. https://doi.org/10.5445/IR/1000162446
[22] Tobias Hey. 2023. Fine-Grained Traceability Link Recovery (FTLR). Zenodo. https://doi.org/10.5281/zenodo.8367392
[23] Tobias Hey, Fei Chen, Sebastian Weigelt, and Walter F. Tichy. 2021. Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2021-09). 12–22. https://doi.org/10.1109/ICSME52107.2021.00008
[24] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. arXiv:1909.09436 (June 2020). https://doi.org/10.48550/arXiv.1909.09436 arXiv:1909.09436 [cs, stat]
[25] Jan Keim, Sophie Corallo, Dominik Fuchß, and Anne Koziolek. 2023. Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery. In *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 141–152. https://doi.org/10.1109/ICSA56044.2023.00021
[26] Jan Keim, Sophie Corallo, Dominik Fuchß, Tobias Hey, Tobias Telge, and Anne Koziolek. 2023. Replication Package for "Recovering Trace Links Between Software Documentation And Code". Zenodo. https://doi.org/10.5281/zenodo.10411853
[27] Yves R. Kirschner, Jan Keim, Nico Peter, and Anne Koziolek. 2023. Automated Reverse Engineering of the Technology-Induced Software System Structure. In *Software Architecture*, Bedir Tekinerdogan, Catia Trubiani, Chouki Tibermacine, Patrizia Scandurra, and Carlos E. Cuesta (Eds.). Springer Nature Switzerland, Cham, 283–291.
[28] P.B. Kruchten. 1995. The 4+1 View Model of architecture. *IEEE Software* 12, 6 (1995), 42–50. https://doi.org/10.1109/52.469759
[29] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. 2015. Can Method Data Dependencies Support the Assessment of Traceability Between Requirements and Source Code? *J. Softw. Evol. Process* 27, 11 (Nov. 2015), 838–866. https://doi.org/10.1002/smr.1736
[30] Michael Langhammer, Arman Shahbazian, Nenad Medvidovic, and Ralf H Reussner. 2016. Automated extraction of rich software models from limited system information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. IEEE, 99–108.
[31] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 324–335. https://doi.org/10.1109/ICSE43902.2021.00040
[32] Sugandha Lohar, Sorawit Amornborvornwong, Andrea Zisman, and Jane Cleland-Huang. 2013. Improving Trace Accuracy Through Data-driven Configuration and Composition of Tracing Features. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 378–388. https://doi.org/10.1145/2491411.2491432
[33] Patrick Mäder and Alexander Egyed. 2015. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering* 20 (2015), 413–441. https://doi.org/10.1007/s10664-014-9314-z
[34] Anas Mahmoud and Nan Niu. 2015. On the Role of Semantics in Automated Requirements Tracing. *Requirements Eng* 20, 3 (Sept. 2015), 281–300. https://doi.org/10.1007/s00766-013-0199-y
[35] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2012. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering* 39, 6 (2012), 869–891.
[36] Andrian Marcus and Jonathan I. Maletic. 2003. Recovering Documentation-to-source-code Traceability Links Using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 125–135. https://dl.acm.org/doi/10.5555/776816.776832
[37] Peter Hugoe Matthews et al. 1981. *Syntax*. Cambridge University Press.
[38] Christoph Mayr-Dorn, Michael Vierhauser, Stefan Bichler, Felix Keplinger, Jane Cleland-Huang, Alexander Egyed, and Thomas Mehofer. 2021. Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1298–1310. https://doi.org/10.1109/ICSE43902.2021.00118
[39] Patrick Mäder and Alexander Egyed. 2012. Assessing the effect of requirements traceability for software maintenance. In *2012 28th IEEE International Conference on Software Maintenance*. https://doi.org/10.1109/ICSM.2012.6405269
[40] Patrick Mäder, Paul L. Jones, Yi Zhang, and Jane Cleland-Huang. 2013. Strategic Traceability for Safety-Critical Projects. *IEEE Software* 30, 3 (2013), 58–66. https://doi.org/10.1109/MS.2013.60
[41] Nenad Medvidovic and Richard N. Taylor. 2010. Software architecture: foundations, theory, and practice. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. 471–472. https://doi.org/10.1145/1810295.1810435
[42] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in Pre-Training Distributed Word Representations. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. European Language Resources Association (ELRA), Miyazaki, Japan. https://aclanthology.org/L18-1008
[43] Chris Mills, Javier Escobar-Avila, Aditya Bhattacharya, Grigoriy Kondyukov, Shayok Chakraborty, and Sonia Haiduc. 2019. Tracing with Less Data: Active Learning for Classification-Based Traceability Link Recovery. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 103–113. https://doi.org/10.1109/ICSME.2019.00020
[44] Kevin Moran, David N. Palacio, Carlos Bernal-Cárdenas, Daniel McCrystal, Denys Poshyvanyk, Chris Shenefiel, and Jeff Johnson. 2020. Improving the Effectiveness of Traceability Link Recovery Using Hierarchical Bayesian Networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 873–885. https://doi.org/10.1145/3377811.3380418
[45] Shiva Nejati, Mehrdad Sabetzadeh, Davide Falessi, Lionel Briand, and Thierry Coq. 2012. A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Information and Software Technology* 54, 6 (2012), 569–590. https://doi.org/10.1016/j.infsof.2012.01.005
[46] Armstrong Nhlabatsi, Yijun Yu, Andrea Zisman, Thein Tun, Niamul Khan, Arosha Bandara, Khaled M. Khan, and Bashar Nuseibeh. 2015. Managing Security Control Assumptions Using Causal Traceability. In *IEEE/ACM 8th SST*. 43–49. https://doi.org/10.1109/SST.2015.14
[47] Kazuki Nishikawa, Hironori Washizaki, Yoshiaki Fukazawa, Keishi Oshima, and Ryota Mibe. 2015. Recovering transitive traceability links among software artifacts. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 576–580. https://doi.org/10.1109/ICSM.2015.7332517
[48] Object Management Group (OMG). 2006. Knowledge Discovery Metamodel (KDM) Specification, Version 1.4. OMG Document Number formal/2016-12 (https://www.omg.org/spec/KDM/1.4/About-KDM/).
[49] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2013. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*. 199–208. https://doi.org/10.1109/CSMR.2013.29
[50] D.L. Parnas. 1994. Software aging. In *Proceedings of 16th International Conference on Software Engineering*. 279–287. https://doi.org/10.1109/ICSE.1994.296790
[51] Cosmina Cristina Rațiu, Wesley K. G. Assunção, Rainer Haas, and Alexander Egyed. 2022. Reactive Links across Multi-Domain Engineering Models. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems* (Montreal, Quebec, Canada) *(MODELS '22)*. ACM, New York, NY, USA, 76–86. https://doi.org/10.1145/3550355.3552446
[52] B. Ramesh and M. Jarke. 2001. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering* 27, 1 (2001), 58–93. https://doi.org/10.1109/32.895989
[53] Michael Rath, David Lo, and Patrick Mäder. 2018. Analyzing Requirements and Traceability Information to Improve Bug Localization. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 442–453. https://doi.org/10.1145/3196398.3196415
[54] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 834–845. https://doi.org/10.1145/3180155.3180207
[55] Patrick Rempel and Parick Mäder. 2017. Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *IEEE Transactions on Software Engineering* 43, 8 (2017). https://doi.org/10.1109/TSE.2016.2622264
[56] Ralf H Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. 2016. *Modeling and simulating software architectures: The Palladio approach*. MIT Press.
[57] Alberto D. Rodriguez, Jane Cleland-Huang, and Davide Falessi. 2021. Leveraging Intermediate Artifacts to Improve Automated Trace Link Retrieval. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 81–92. https://doi.org/10.1109/ICSME52107.2021.00014
[58] Per Runeson and Martin Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. 14, 2 (2008), 131. https://doi.org/10.1007/s10664-008-9102-8

[59] Aaron Schlutter and Andreas Vogelsang. 2021. Improving Trace Link Recovery Using Semantic Relation Graphs and Spreading Activation. In *Requirements Engineering: Foundation for Software Quality*, Fabiano Dalpiaz and Paola Spoletini (Eds.). Springer International Publishing, Cham, 37–53.

[60] Aaron Schlutter and Andreas Vogelsang. 2021. Improving Trace Link Recovery Using Semantic Relation Graphs and Spreading Activation. In *Requirements Engineering: Foundation for Software Quality*, Fabiano Dalpiaz and Paola Spoletini (Eds.). Springer International Publishing, Cham, 37–53. https://doi.org/10.1007/978-3-030-73128-1_3

[61] Marcelo Schmitt Laser, Nenad Medvidovic, Duc Minh Le, and Joshua Garcia. 2020. ARCADE: an extensible workbench for architecture recovery, change, and decay evaluation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1546–1550.

[62] Kamal Souali, Othmane Rahmaoui, and Mohammed Ouzzif. 2016. An overview of traceability: Definitions and techniques. In *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*. 789–793. https://doi.org/10.1109/CIST.2016.7804995

[63] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. 2022. Relationships between software architecture and source code in practice: An exploratory survey and interview. *Information and Software Technology* 141 (2022), 106705.

[64] Ryosuke Tsuchiya, Kazuki Nishikawa, Hironori Washizaki, Yoshiaki Fukazawa, Yuya Shinohara, Keishi Oshima, and Ryota Mibe. 2019. Recovering transitive traceability links among various software artifacts for developers. *IEICE TRANSACTIONS on Information and Systems* 102, 9 (2019), 1750–1760. https://search.ieice.org/bin/summary.php?id=e102-d_9_1750

[65] Zhiyuan Wan, Yun Zhang, Xin Xia, Yi Jiang, and David Lo. 2023. Software Architecture in Practice: Challenges and Opportunities. *arXiv preprint arXiv:2308.09978* (2023). accepted for ESEC/FSE 2023.

[66] W. Wang, N. Niu, H. Liu, and Z. Niu. 2018. Enhancing Automated Requirements Traceability by Resolving Polysemy. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*. 40–51. https://doi.org/10.1109/RE.2018.00-53

[67] Meng Zhang, Chuanqi Tao, Hongjing Guo, and Zhiqiu Huang. 2021. Recovering Semantic Traceability between Requirements and Source Code Using Feature Representation Techniques. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 873–882. https://doi.org/10.1109/QRS54544.2021.00096

[68] T. Zhao, Q. Cao, and Q. Sun. 2017. An Improved Approach to Traceability Recovery Based on Word Embeddings. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 81–89. https://doi.org/10.1109/APSEC.2017.14