Karlsruhe Institute of Technology

Linking Software Architecture Documentation and Models

Master Thesis of

Sophie Schulz

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Anne Koziolek
Second reviewer:	Prof. Dr. Ralf Reussner
Advisor:	M.Sc. Jan Keim
Second advisor:	Dipl.-Inform. Angelika Kaplan

6. April 2020 – 5. October 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 5. October 2020

. . . . . .

(Sophie Schulz)

Abstract

In software engineering, consistency between artifacts is an important topic. This thesis proposes a structure for detecting mutual and missing elements between a documentation and a formal model.

First, the approach identifies and extracts model instances and relations described in the text. Then, the approach connects these textual elements to their corresponding counterparts in the model. These links are comparable to trace links. However, the approach allows the graduation of these links. Moreover, the approach generates recommendations for elements that could not be found in the model.

The approach identifies model names and types with an F1-Score above 54%. 60% of the recommended instances match the instances found in the user study. For the identification of relations and the creation of links the approach achieved promising results. The results can be improved through future work. This can be realized because the design allows an easy implementation of extensions.

Zusammenfassung

In der Softwareentwicklung ist die Konsistenz zwischen Artefakten ein wichtiges Thema. Diese Arbeit schlägt eine Struktur zur Erkennung von korrespondierenden und fehlenden Elementen zwischen einer Dokumentation und einem formalen Modell vor.

Zunächst identifiziert und extrahiert der Ansatz die im Text beschriebenen Modellinstanzen und -beziehungen. Dann verbindet der Ansatz diese Textelemente mit ihren entsprechenden Gegenstücken im Modell. Diese Verknüpfungen sind mit TraceLinks vergleichbar. Der Ansatz erlaubt jedoch die Abstufung dieser Links. Darüber hinaus generiert der Ansatz Empfehlungen für Elemente, die nicht im Modell enthalten sind.

Der Ansatz identifiziert Modellnamen und -typen mit einem F1-Score von über 54%. 60% der empfohlenen Instanzen stimmen mit den in der Benutzerstudie gefundenen Instanzen überein. Bei der Identifizierung von Beziehungen und dem Erstellen von Verknüpfungen erzielte der Ansatz vielversprechende Ergebnisse. Die Ergebnisse können durch zukünftige Arbeiten verbessert werden. Dies ist realisierbar da der Entwurf eine einfache Erweiterung des Ansatzes erlaubt.

Contents

Abstract i

Zusammenfassung iii

1. Introduction 1

2. Related Work 5
2.1. Information Retrieval . . . 5
2.2. Generating Artifacts . . . 7
2.3. Finding Trace Links . . . 9
2.4. Consistency Tests . . . 10

3. Fundamentals 11
3.1. Palladio Component Model . . . 11
3.2. PARSE And INDIRECT . . . 12

4. Approach 13
4.1. Overview . . . 13
4.1.1. Example 1: Introduction To Hint And Trace Links . . . 14
4.2. Detailed Approach . . . 16
4.2.1. Text Extraction . . . 17
4.2.2. Model Extraction . . . 20
4.2.3. Recommendation Generation . . . 21
4.2.4. Link Generation . . . 23

5. Architecture 27
5.1. High-level Architecture . . . 27
5.1.1. Analyzers And Solvers . . . 28
5.2. Detailed Architecture . . . 29
5.2.1. Text Extractor . . . 29
5.2.2. Model Extractor . . . 32
5.2.3. Recommendation Generator . . . 33
5.2.4. Connection Generator . . . 34

6. Implementation 37
6.1. Text Extractor . . . 37
6.1.1. The Adding Of Basic NounMappings . . . 38
6.1.2. The Adding Of NounMappings Containing Separators . . . 41
6.1.3. Similarity . . . 42

6.1.4. Adding Term Mappings And Relation Mappings 42
6.1.5. Analyzers And Solvers 42
6.2. Recommendation Generator 46
6.2.1. Analyzers And Solvers 48
6.3. Connection Generator 53
6.3.1. Analyzers And Solvers 54
6.4. Configuration 55

7. Evaluation **57**
7.1. Representing Links 57
7.2. Creation Of A Gold Standard For Evaluation 62
7.2.1. Preparation 62
7.2.2. Concept Of The User Study 64
7.2.3. Execution Of The User Study 66
7.2.4. Creation Of The Gold Standard 66
7.3. Metrics 68
7.4. Textual Element Recognition 69
7.5. Model Element Recognition 71
7.6. Creating Tracelinks 73
7.7. Threats To Validity 75

8. Conclusion And Future Work **77**

Bibliography **81**

A. Appendix **85**
A.1. Implementation 85
A.2. Evaluation 89

List of Figures

1.1. Model Extraction Overview: The model is represented as internal structure. Instances consist of names and types. Relations have participating instances and a type. . . . 2
1.2. Text Extraction: Names, types and relations are marked in the text. Then instances and relations are built out of them. . . . 3
1.3. Approach Overview: The extracted model and the textual model are joined to a version that is recommended to the user. . . . 4

3.1. An example of a composite component in PCM. The composite component contains two basic components. . . . 11
3.2. Graph structure of PARSE: The tokens of the input text are represented by nodes and connected with *next edges*. The graph can be extended with additional edges and tags. . . . 12

4.1. Approach Overview . . . 14
4.2. The transformation of a model instance to a goal model instance. The model instance is missing an element . . . 15
4.3. The approach in levels: The text extraction and the model extraction are the base of the approach. The recommendations are built upon textual and model information. The creation of links depends on the information of recommendations and the model. . . . 18
4.4. Example 2: The initial input text . . . 18
4.5. Example 3: The initial input model . . . 20

5.1. High-level Architecture: The arrows represent the requirements of each level. . . . 27
5.2. Analyzer Architecture . . . 29
5.3. Solver Architecture . . . 30
5.4. Text Extractor Architecture . . . 31
5.5. Model Extractor Architecture . . . 32
5.6. Recommendation Generator Architecture . . . 33
5.7. Connection Generator Architecture . . . 34
5.8. Architectural relationships between all states . . . 35

6.1. Overview of the Text Extraction State as UML class . . . 37
6.2. NounMapping as UML class . . . 38
6.3. The analysis of a sentence with a dependency parser. . . . 44
6.4. Recommendation Generator . . . 47
6.5. Connection State . . . 54

7.1. Approach Overview 58
7.2. Diagram of Teammates used for the evaluation [29]. 63
7.3. Model of a confusion matrix 68

List of Tables

4.1. Example 1: An example of trace links . . . 15
4.2. Example 1: The resulting hint links . . . 16
4.3. Example 2: The results of the noun classification . . . 19
4.4. Example 2: The extracted relations of the text . . . 20
4.5. Example 3: The extracted instances of the model . . . 21
4.6. Example 3: The extracted relations of the model . . . 21
4.7. Example 4: The updated names, types and norts of the text extraction . . . 22
4.8. Example 4: The updated relations of the text extraction . . . 22
4.9. Example 4: The recommended instances . . . 23
4.10. Example 4: The recommended relations . . . 23
4.11. Example 5: The instance links . . . 24
4.12. Example 5: The relation links . . . 24

6.1. The mapping of incoming dependency tags to concluded noun mapping kinds . . . 45
6.2. The mapping of outgoing dependency tags to concluded noun mapping kinds . . . 45

7.1. The instances of the TEAMMATES diagram in the model extraction state. 62
7.2. The relations of the TEAMMATES diagram in the model extraction state. 62
7.3. The instance table of the first task of the user study. . . . 65
7.4. The relation table of the second task of the user study. . . . 65
7.5. The results of the evaluation of the marks of the first task in percent per reference sets . . . 70
7.6. The results of the evaluation of the instances of the first task in percent by matching decisions. Only references with more than four points were considered. . . . 71
7.7. The results of the evaluation of the tags of the third task in percent per reference sets . . . 71
7.8. The results of the evaluation of the instances of the third task in percent by matching decisions. . . . 72
7.9. The results of the evaluation of the recommended relations compared to the existing *in* relations of the model, in percent by matching decisions. . 73
7.10. The results of the evaluation of the instances of the third task in percent per matching decisions. . . . 74
7.11. The results of the evaluation of the recommended relations compared to the existing *in* relations of the model, in percent per matching decisions. 74

A.1. The general configurations that can be set in the configuration file. . . . 85

A.2. The configurations that can be set for the connection generator. 85
A.3. The configurations for helper classes, such as *SimilarityUtils* 86
A.4. The configurations of all parts of the text extraction level. 87
A.5. The configurations that can be set for the recommendation generator. . . 88

1. Introduction

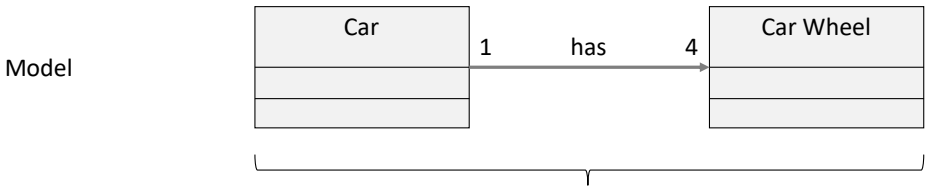
In software development, communication is an important topic: If requirements are misunderstood the client most often will not get the software as ordered; if the architecture is not documented well, further development and maintenance can get complicated and exhausting. With clear, describing, explaining documentations, software processes can improve. However, this improvement comes at a cost. When describing software under development in documents, these have to be adapted to every change on software side. For this, similar to programming without an IDE (marking all faults), mistakes are almost unavoidable: Formulations will be unclear, old references will be forgotten, or new functions are not described. Thereby, it is difficult for readers to connect textual elements to instances of the described model. These connections are called trace links. In general, the concept of traceability is based on the assumption that document and model are consistent. Thus, a trace link always has two end points: One in the model and one in the document. These trace links are created when reading a document with the model in mind. If document and model can not be connected, the reader searches for a missing detail in the model or claim the description of the document as faulty. In case of designing a more efficient procedure (than looking manually for faults) first steps have been made to automate consistency checks between documents and underlying models. Thus, faults can be marked and are corrected immediately (like in an IDE). This automation is not easy, as I will show in Chapter 2, especially when the end points are ambiguous.

In this work, a consistency analysis between architecture documentation and model is processed. Faults, like forgotten descriptions, are recognized. For this reason, I introduce hint links, a graduation of trace links. Hint links can be seen as potential trace links. Instead of having approved endpoints, they can mark a textual element which seems likely to be a description of a model instance. In contrast to the common understanding of trace links, the textual element has not to be connected to a model element. Therefore, hint links can be used similar to loose trace links, for example if a textual element could be identified as ambiguous. With hint links the text element could be mapped to multiple model elements. Thereby, neither the connection is discarded, nor are all proposed to the user, but the best suggestions can be proposed to the user in order of their certainty. Moreover, in contrast to trace links, hint links are able to mark textually described elements that possibly should be contained in the model, even if they are not contained. Thus, with the help of trace and hint links the reader is supported, faults can be found efficiently, and misunderstandings can be avoided.

This work finds trace links and hint links between architecture documentations and models. In this master thesis, I use INDIRECT (see Section 3.2) for the textual preprocessing of the documentation. Thereby, the textual information is transmitted in a graph and become machine-readable. In the graph, every node represents a token. During preprocessing INDIRECT adds additional linguistic information as edges or annotations.

This work divides model elements in instances and relations. Every instance is defined by a name and a type. Every relation consists of at least two instances and a type. The approach uses this definition to get first indications what parts of the text describe model instances or relations.

Figure 1.1 shows a short application of the definition and its meaning. In this example, a simple UML class diagram is given. The instances of the model are represented as a name-type combination. The type is the type of the model element. In this case, both types are *Class*. The model has only one relation. Relations are represented with their instances and a type. Here, the instance are referenced with the name. The type is (similar to the type of instances) the type of the chosen model element. In this case, the model element is an association that is translated to *has*. Thereby, the model is represented textually in the table.



Internal Model Representation

Name	Type
Car	Class
Car Wheel	Class

Instance 1	Instance 2	Type
Car	Car Wheel	association

Figure 1.1.: Model Extraction Overview: The model is represented as internal structure. Instances consist of names and types. Relations have participating instances and a type.

The idea of the approach is to extract information from the text that indicate instances or relations. Figure 1.2 shows this procedure. The approach starts at the textual base and searches for types and names. At first, this is done without knowledge of the model. This generalized search allows finding names and types of elements that are missed in the model. In the example, the process is shown with the marks in the documentation text. The process identifies *Car*, *Car Wheel*, *Driver*, and *Automobile Model* as names. Moreover, the approach recognizes *Classes* as type, and the relations *from* and *owner of*.

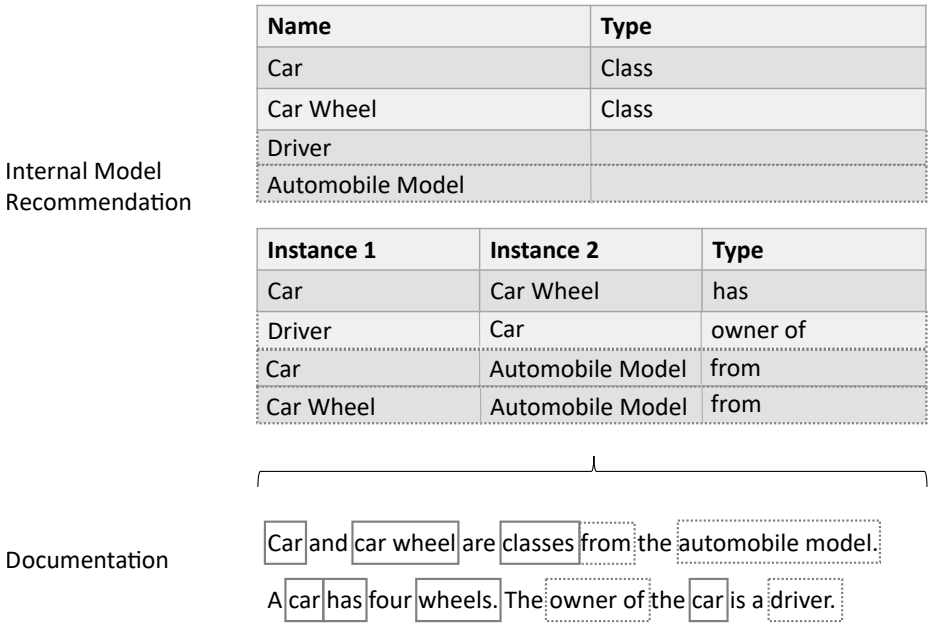


Figure 1.2.: Text Extraction: Names, types and relations are marked in the text. Then instances and relations are built out of them.

Based on these marks the approach tries to build possible instances and relations. For this purpose, the model can be used as background knowledge. Hint links and trace links are only created in the step after this. Thus, the approach currently uses the model only as support for the search of instances and relations. With the help of the model, the approach can use the occurring types for the identification of instances. In the example, the approach could thereby recognize the name-type combinations of *car - class* and *car wheel - class*, but this can also be done without model knowledge. The increased certainty by the verification of the model is shown with the solid lines. Less confident elements are marked with dotted lines. For the other found names the model does not contain a hint. Neither *driver* nor *automobile model* have a connection to a type. Therefore, their generated recommendations contain only a name. For the relation this process is done in a similar manner. Instead of searching for the same type as stored in the model, the approach searches for similar participating instances. *Car* and *Car Wheel* are identified as possible instances. Thus, relations with them should be in the model. All other relations are also transmitted in the model.

Through the previous steps, the approach has two versions of a model. These versions have to be joined to a recommended version. At this point, hint links and trace links are created. Figure 1.3 shows a merged version of both models. Trace links are represented with solid lines while the dotted lines represent hint links. In this example, the approach decides that driver is a missing element of the model. Since the model type could not be identified, the relation type stays at its textual description. Thereby, the user can better understand the meaning of the relationship.

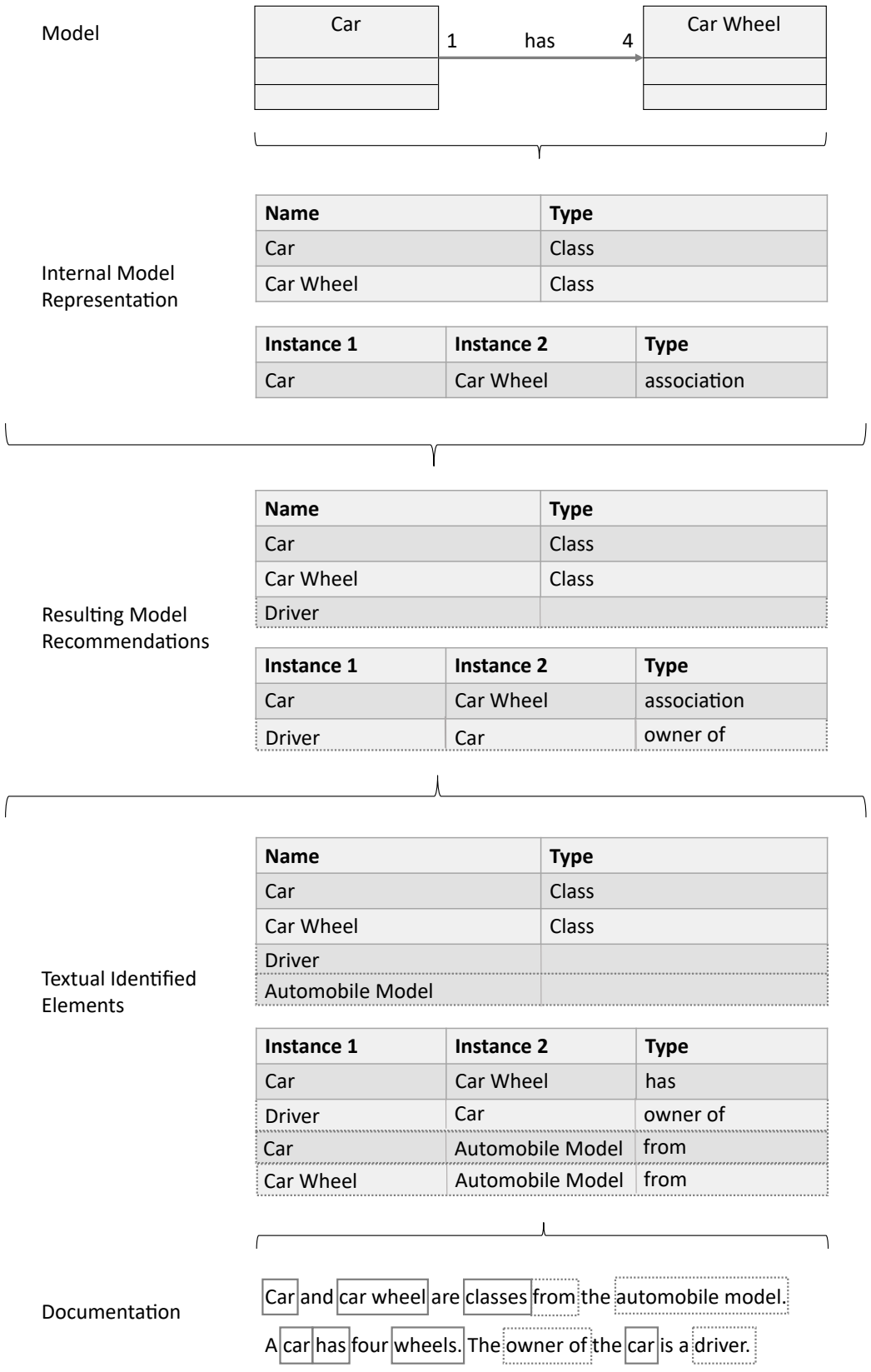


Figure 1.3.: Approach Overview: The extracted model and the textual model are joined to a version that is recommended to the user.

2. Related Work

This chapter presents work to related topics of this thesis. The first step of this thesis is the extraction of textual information. This extracted information is used to generate hint and trace links. In general, this subject is called information retrieval and is introduced in Section 2.1. The next sections present different view points of the problem. In this thesis, text and model are present and should be connected. Both, the text and the model can contain inconsistencies to the goal model. The goal model represents the current state of a software that is described by the text and model. Instead of generating the goal model, the trace links (representing consistencies), hint links (representing possible consistencies), and inconsistencies are suggested to the user. In this thesis neither the textual input nor the model input are assumed to be correct. In contrast to this assumption, the related work presented in Section 2.2 and Section 2.3 require correct and therefore consistent resources. While Section 2.2 focuses on the generation of artifacts to a given meta model and textual description, Section 2.3 focuses more on the actual connection between the artifacts. In these sections, the assumption that all artifacts are correct is crucial. Thus, missing elements etc. can not be found. Finally, Section 2.4 presents work that does not require consistent artifacts. This section focuses on different consistency testing approaches and includes thereby related concepts to hint links.

2.1. Information Retrieval

One of the first objectives of this work (see also Chapter 1), is to find and mark names and types of potential goal elements in a document. For consistency analysis, this has to work without any knowledge of the model. Thus, in a first step all potentially relevant information of a text have to be identified and extracted. This section introdurces different approaches for information retrieval.

In the 1990s, the interest in information retrieval started to grow. One of the first goals was to find conceptual related terms. In 1992, Hearst [13] started to search for hyponymous entities in corpora using predefined patterns. In these patterns, syntactical elements are used as place holders for sub terms and generic terms. The connections between them are natural formulations. By simple pattern matching with "NP_x and other NP_y" phrases like "... temples and other civic buildings ..." confirm the term NP_x as a hyponym of NP_y.

A related approach is DIPRE [5] using Bootstrapping. To find entities to given relations (e.g. author - book) a database is searched for linguistic formulation between given start terms. These connections are extracted and used as formulations in hearst patterns to find more entities, similar to the given terms (in this example authors and books). On the one hand, this approach improves the applicability of hearst patterns a lot by automatize the generation of the patterns. On the other hand, the approach turned out as very dependent

of the start terms. In the evaluation, author-book-pairs were searched, starting with five pairs. After the evaluation, it was found out that only the first two pairs had generated the search patterns. Therefore, and because both books were of the science fiction genre, the new find instances mostly were of this genre, too. Even though this approach has its weaknesses, the idea of automated relation extraction for finding entities is kept.

KnowItAll [11, 10] was build upon this bootstrapping principle. As in the related work above, the goal is to extract concepts and relationships. To find new instances of conceptual classes of entities, hearst patterns are used. Then, a search similar to the one in DIPRE is used, to find more formulations of relations. By this, with just two start values (e.g. *Karlsruhe* and *Germany*), a *capital_of*-relation can be instantiated. After extracting relations, these are combined with related entities and entered in a search engine. A score (pointwise mutual index) is calculated with the count of results. Finally, a naive-bayes-classifier is trained with this score to verify candidates. Thereby, KnowItAll does not need manual input and is classified as an unsupervised learner.

TextRunner [2] optimizes the extraction of relations and entities, being an unsupervised single pass extractor. Single pass extractors only need to run one time through the input data. Especially when using larger data sets, as in entity and relation extraction, this is a huge improvement of efficiency. TextRunner extracts all relations of a small, unlabeled training set. These are stored as triples like $relation = (NP_x, relation_{xy}, NP_y)$. The relation is marked as a positive sample, if a short dependency path exists between both nominal phrases, none of them is a pronoun, and no sentence limit is crossed. Otherwise, the relation is marked as a negative sample. With these marked data a naive bayes classifier is trained. In a second step a corpus is passed through, tagging type of words, identifying nominal phrases, and extracting them, as well as potential relations. After applying multiple heuristics (e.g. removing adverbs and prepositions), triples of relations are generated from the extracted data. Then, the classifier rates the new triples. As KnowItAll, TextRunner includes a step evaluating its results. In this case, this is done by the princile of the urn model: For every relation its occurrences are counted and added to the urn model. The probability that a relation is correct, is the probability of drawing a ball of this relation. A disadvantage of this last step is that the account of relations sometimes is very independent of its correctness. In this case, when using the internet as a corpus, curios relations, like "Chuck Norris, invented, the internet" were marked as correct. Obviously this problem is caused by using unrestricted input data. In general, the evaluation of this approach included 60.5 million relations, extracted from 9 million websites and 133 million sentences. By evaluating a sample of 400 of these, only 35 were well-formed (e.g. not (29, dropped, instruments)) and explicit (e.g. not (Einstein, derived, theory)). However, of these explicit, well-formed triples 88.1% were correct.

Distant Supervision [21] invented a new approach to avoid not well-formed relations and entities. It is based on the paradigm: If two relation-connected entities occur in a sentence, then this sentence represents the relation. Similar to Bootstrapping in this approach predefined instances are necessary. To be less susceptible than Bootstrapping more start values are used. As resource corpora like WordNet can be used, to extract entity pairs of listed relations (e.g. (lion, animal) from the *is-a* relation). For these pairs occurrences in another corpora (e.g. Wikipedia) are used to extract lexical, syntactical, and naming properties for relations between them. Finally, these properties are used in

a classifier for this relation. Thereby, search patterns for e.g. (hyponym,hyperonym) - relations can be found and help to extract more entities and relations representing their relationship.

The information retrieval is related to one of the first tasks of this thesis: The identification and extraction of model elements described in the documentation. Therefore, similar pattern based approaches would be applicable. As Chapter 4 will describe, the approach will search for names and types in the text. The classification of them is done with simple patterns. In contrast to the work presented in this section, the approach uses the identified textual information to combine and aggregate it to model elements.

2.2. Generating Artifacts

This section provides multiple approaches for extracting textual information and transcribing them in a given meta model, like UML models. This goal is related to the creation of hint links. An idea would be to propose a trace link or hint link, instead of creating an element of a model.

Similar to the information extraction (Section 2.1) the approaches of this section are rule-based. In contrast to the firstly described approaches, the approaches of this section transfer the found textual information in a meta model. After different textual preprocessing steps, clauses are checked and parts of the sentences are instantiated as model elements. Some examples for these approaches are the approach by Deeptimahanti & Babar [9] and another approach by Ibrahim & Ahmad [15]

In [9] requirement documentations are analyzed to derive UML models such as use case or class diagrams. For this purpose, Deeptimahanti and Babar use a rule-based approach. A linguistic analysis using the Stanford parser, WordNet, and JavaRAP extracts lexical and syntactical properties. Based on these properties and pre-defined rules, the models are generated. The rules do not seem to be very complex. When generating a design diagram, structures such as “Who does what to whom?” are used. The subjects and objects are derived from the nominal phrases and the predicate from the verb of the input sentence.

A similar approach is pursued in [15]. In this approach (as in [9]), a linguistic analysis is first carried out and a class diagram is created from requirement texts. For this purpose, Ibrahim and Ahmad use OpenNLP and WordNet. After applying the different analysis steps, the extracted concepts are examined. According to the concepts their approach denotes rules. These are then compared using an ontology and classified as elements of the class diagram based on additional, more complex heuristic rules. In this way classes, attributes, and various relations are recognized.

As described in Section 2.1, one of the problems of these rule based approaches is the worse classification when rules are manually created.

Instead of manually creating rules for classifying model elements, sometimes information is first transferred in an intermediate storage. Therefore, when analysing text information, these storages are used to make the extracted information accessible to other processes. In computer linguistics, these storage structures are often termed as ontologies. The term ontology references to the philosophical term for doctrine of being [8]. In computer linguistics, the term indicates a storage structure providing as much shared

explicit understanding as possible. If possible, not only facutal knowledge, like ("There are ingredient on a pizza"), but also conceptual knowledge (e.g. in which order a pizza is topped) is stored. In order not to get lost in unnecessary details, ontologies are often restricted to the domain of the application system. In general, ontologies consist of concepts, instances that inherits from concepts, and relations connecting concepts and classes. Sometimes partitions are used to divide concepts into different sub-concepts (e.g. animals in mammals and other).

There are some approaches for the automatic generation of ontologies. Usually, the creation of an ontology includes the identification of problem-relevant elements, concepts and instances, the determination of concept attributes, and insertion of relations between the concepts.

Kof pursues a syntactic approach to this in his work [16], [17], [17], [18]. The approach aims at the generation of an ontology from requirements documents or specification. For this purpose, the text predicates and their arguments are extracted. The result of the extraction are terms, i.e., subjects and objects that are linked by their connecting verbs. For the subsequent clustering process, the verbs are first brought into their basic form. In order to arrange these clusters that consist of nouns connected by a predicate taxonomically, manual effort is necessary. If the taxonomy exists, relations must be searched for. For this, association rules are used. For Kof, these rules are formed using words within a sentence. The resulting relations are suggested to the user. Thus, in this approach a manual correction of suggestions and a comprehensible naming of the relations is included in the ontology. After the development, the approach is tested in a case study. This is based on a six-page specification description of a steam boiler and the corresponding software. In this first case study, Kof has major problems with the structure of the specification: Sentences that contain only incomplete information as well as enumerated sentences cause problems for term extraction. The exact specification of nouns also plays an important role: According to this, states should get meaningful names and always be addressed via these names. After the text is improved and an ontology has been created, the absence of two concepts becomes apparent during the evaluation. These were not extracted above, since they are only used once in the specification text and are unconnected to the rest of the system. It is therefore unclear whether this should be interpreted as a weakness of the system, as the concepts (if relevant) should have been described more precisely. Since associations and properties of the concepts are entered manually, a further completeness analysis is not necessary. For Kof, however, the creation of an ontology is only an aid to derive a system model manually. By combining the ontology with a meta-model for distributed systems, components, their behavior and communication channels are extracted. However, the problem arises that although superordinate concepts can be considered as components, the taxonomy is much less helpful in identifying the communication channels. One reason for this is that the ontology itself does not reflect the meta model. In a second case study the scalability of ontology creation is examined. For an 80 page specification of an instrument cluster for a car, a total of five working days of manual work is required. First, the document is made readable on the first day, then 1.5 days of manual work are spent correcting errors in the document. In particular, this correction includes lists and tables, but also the completion of key sentences and grammatical improvements. This step also promotes the specification, which is controlled and improved in this way. After term

extraction, 1.5 days are needed to arrange the cluster taxonomically. The remaining day is used to insert found associations as relations in the ontology. Thus, with an acceptable manual effort, the creation of a specification reflecting ontology, as well as an improvement of the specification, has taken place.

The approach of Kof shows the complexity and importance of a correct information extraction. Moreover, it shows that creating a model out of textual information is not a trivial goal. Textual based created models could be used to generate trace links and hint links, by comparing the created and given artifact. This approach is treated in the following Section 2.3.

The approaches of this section all create artifacts from a textual resource. This is related to the derivation of model elements in this thesis. In contrast to the presented approaches, this thesis compares the identified model elements to an existing model. Then consistencies and inconsistencies are found. Finally, the user gets suggestions for model elements. Therefore, in this thesis no unambiguous model is derived by the text.

2.3. Finding Trace Links

In this section, multiple approaches for trace link identification are introduced. The focus lies on identification and extraction of verified connections between text and another artifact. In this context, every trace link has two approved end points. Consistency tests, including hints for missing model elements etc., are more special and presented in Section 2.4.

The work of [28] is based on collecting links between requirements and architecture. They define an ontology in which specifications and architectural artifacts can be defined manually. The latter are documented in a semantic wiki. This can be used to find and point out architectural conclusions about given requirements.

An approach, which tries to determine these connections automatically, is presented in [22]. Here, requirements are written in user stories and epic stories. With the tool RE4SA these are assigned to their architectural counterparts. Two processes are presented: At first, a top-down process “Architecture Discovery”, which creates an architecture based on requirements and subsequently connects modules to epic stories and features within the modules to user stories. At second, a complementary bottom-up process “Architecture Recovery” is used to restore the architecture from an implemented system. This is done using available documentation such as source code, a version of the system. The recovered components are then assigned to the requirements.

Guo et al.provide an approach for the automatic insertion of trace links between requirement texts and other artifacts using a Recurrent Neural Network (RNN) [12]. For this, contexts of words by learning word vectors are considered. For each artifact (requirement texts, source code files, etc.) each word is replaced by the associated, already learned vector. These vectors are then entered sequentially into the RNN. The output is a vector representing the semantic information of the entire artifact. Finally, the tracing network compares the semantic vectors of the different artifacts and calculates a probability for the connection. The approach is evaluated using a communication-based control system for trains. This system ensures that trains stay on their scheduled routes to avoid accidents.

To train the RNN, texts from Wikipedia, the domain, and software artifacts are used. The resulting data set includes over 1500 software subsystem requirements (SSRS), as well as about 500 Software Subsystem Design Descriptions (SSDDs). During training, the SSRSs are used as source artifacts and the SSDDs as target artifacts. Unfortunately, such a large amount of data is usually not available. Therefore, this work is only applicable in rare cases.

2.4. Consistency Tests

In this section, the related work provides insights in current consistency test strategies. In contrast to the work in Section 2.3, the work of this section is able to identify possible failures between the compared artifacts. The first approach introduced presents an approach for consistency testing between different models. Thereby, it gives an idea of how to reach consistency between predefined models. The latter work in this section is much more related to this work. It provides consistency tests by using a controlled language for architecture description.

Vitruvius [7, 6, 19] is an approach for consistency assurance between different meta-models. The idea behind Vitruvius is that software can be viewed from different perspectives. These views contain the information of several underlying meta-models. Instead of using a single underlying model (SUM), Vitruvius defines transformations between the meta-models to achieve consistence between them.

In [25, 26, 27] Schröder et al. provide an approach for conformance checking between source code, architecture, and architecture documentation. The architecture concepts are formalized in a controlled language. It allows them to express and connect architecture rules directly with project-specific concepts. Therefore, the architecture concept language must reflect the architecture concepts, relations and rules. The source code is used and represented as an ontology, whereas an architecture-to-code-mapping already exists. While the concept language and architecture-code mapping have to be provided by an architect, the source code ontology can be automatically generated. Then, the architect has to create a project-specific architecture ontology using the architecture documentation in the controlled language. Then, the approach extracts architecture concepts from the code, by using the architecture-code mapping to gain the architecture. Finally, the extracted architecture is compared to the architecture rules, defined by the architect.

In contrast to the approaches presented in this section, this thesis will provide an approach for consistency testing between natural language and a model. Unlike Schröder et al. neither the language should be restricted, nor the rules predefined by a software architect.

3. Fundamentals

This work is based on architecture models and documentations of software. In Section 3.1, this chapter provides a short overview of architecture models and describes what kind of models are used. Furthermore, the approach uses PARSE and INDIRECT as preprocessing steps for the documentation. By the application of these, linguistic information is annotated to the text. Both are described in Section 3.2.

3.1. Palladio Component Model

In general, the software architecture describes the fundamental concepts of a system and its environment. It contains the elements and relations of the system and represents its design. Nevertheless, the architecture model is a reduced variation of the reality. Thereby, the architecture model is simpler to understand and overview than the architecture in all its details. The model is often used for a better communication and analysis of the system. Architecture models can be expressed in different kind of models.

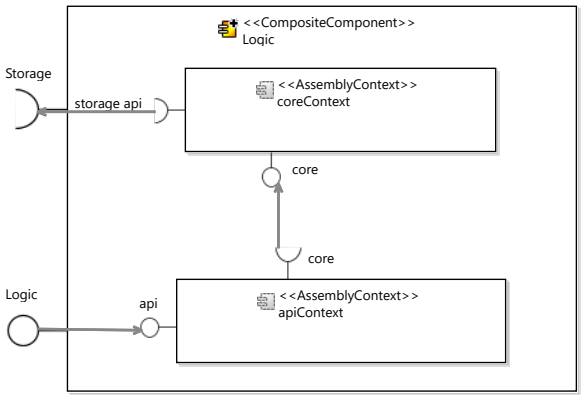


Figure 3.1.: An example of a composite component in PCM. The composite component contains two basic components.

The approach of this thesis is independent of the kind of the underlying architecture model. For the evaluation, the palladio component model is used. To create different views, the Palladio Component Model has been developed to represent a software architecture (see [4]). It offers the possibility to reflect the architecture of an existing software system [3]. With the help of the meta-model, components can be specified, connected and their behavior can be described. A software component can be described as a "contractually specified building block for software, which can be composed, deployed, and adapted without understanding its internals" [24]. Thereby, components encapsulate parts of the

system. Every component has an provided interface that describes the service provided by the component, as well as a required interface that specifies the needs of the component. Palladio differs between two kinds of components: Basic components and composite components that encapsulates multiple basic components.

3.2. PARSE And INDIRECT

PARSE (Programming Architecture for Spoken Explanations) is a toolkit for agent-based processing of natural language [31]. For the processing within PARSE the textual input is stored in a graph structure. Actually, PARSE is a tool for spoken language. To transform the spoken input to text, an automatic speech recognition is used. Since this procedure does not recognize punctuation marks PARSE offers only a shallow natural processing. Each processing is done via an agent. Agents are graph editors with a specific task (e.g. to add part of speech tags). Thereby, the PARSE graph becomes to a knowledge base of linguistic information annotated to the textual input. Figure 3.2 gives an overview how a PARSE graph could look like. PARSE transforms the given text in a graph. Each node represents a token. The order of the tokens within the sentence is represented as *next* edges. PARSE agents are able to extend the graph by additional edges, tags or new nodes. In this case, part of speech tags and dependency edges were added. PARSE offers a lot more agents that can be applied to the graph.

Instead of PARSE, this work uses INDIRECT (Intent-driven Requirements-to-Code Tracability) [14]. It complements PARSE by processing textual input (including punctuation marks). In this work, only this deviating property is used. However, in future work, the advantage of opportunities offered by INDIRECT could be taken.

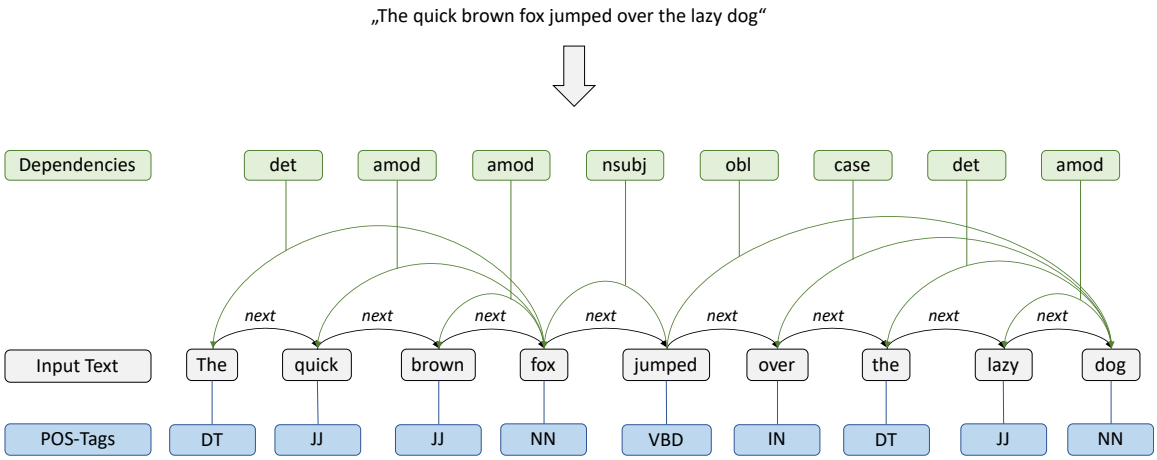


Figure 3.2.: Graph structure of PARSE: The tokens of the input text are represented by nodes and connected with *next edges*. The graph can be extended with additional edges and tags.

4. Approach

The goal of this thesis is to connect architecture documentation and models while identifying missing or deviating elements. An element can be any representable item of the model, like a component or a relation. Section 4.1 provides an oversight of the whole approach. Thereby, the concepts of hint links and trace links are introduced. In Section 4.2, the approach is explained in more detail, including the steps of extraction, hint link and trace link creation.

4.1. Overview

Figure 4.1 presents this approach. First, all needed information from documentation and the architecture model are extracted, to be independent of any architecture style and underlying meta-models. Thus, any kind of models and texts could be used. Model elements are currently defined as instances and relations. Instances are represented by a name-type combination (e.g. *Car-Class*). Relations consist of a type and participating instances (e.g. *has-Car-Car Wheel*). The goal is to connect similar text elements and model elements and to identify missed ones.

Figure 4.2 shows that these connections can be different. The solid connection between text elements and model elements represents the certainty that both represent the same instance. This connection represents trace links. Above the solid line is a dotted line. This line represents hint links. Whenever the certainty for the similarity of both element is too low for a trace link or the endpoints are ambiguous, the connection is called a hint link. With the exclusion of possibilities or an increase of the certainty, this hint link could become a trace link. The arrows to the goal model represent the suggestions that are presented to the user. The goal model is the model that represents the current state of the software. In this thesis, I assume that all in the text consistently described model elements are part of the goal model. Therefore, all trace links that connect the model element and their textual description are claimed to be in the model. Since these elements are described in both, the model and the text are consistent. Hint links could also be suggested to a user. By verifying a hint link, a trace link would be created and the described element would be assumed to be in the model. Furthermore, model elements that are not contained by links are suggested to be in the goal model. Since they are not described in the text, they represent an inconsistency. Also, text elements can be suggested to be part of the goal model. Both of these suggestions could be seen as further hint links. Since the approach assumes that the model is correct and the text is not, in the following only the suggestion at the left side of the figure is also called hint link. The definitions for hint and trace links can be summarized as:

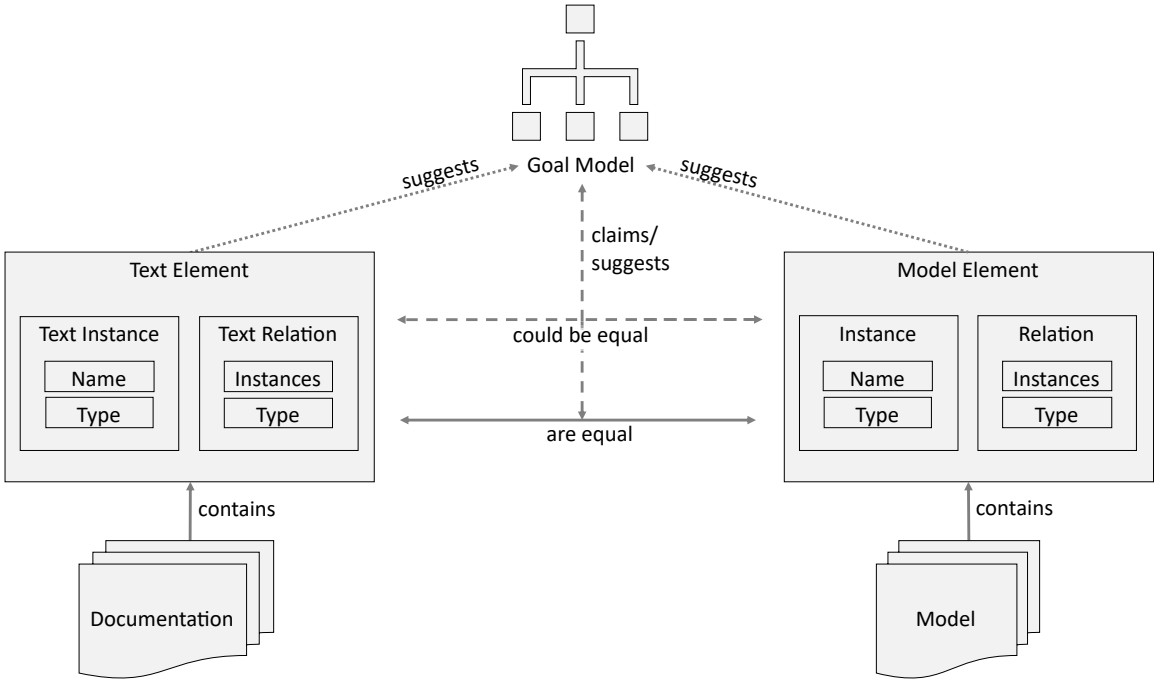


Figure 4.1.: Approach Overview

Hint Links
: are hints for elements to be included in the correct model. A hint can be a single element, or an element found in both the initial model and text. Every hint link has a probability that represents the certainty that the endpoints are equal.

Trace Links
: are hint links, with two end points and a high probability (that these endpoints are equal). This means that the end points of the link are almost surely the same instances.

4.1.1. Example 1: Introduction To Hint And Trace Links

This example highlights the problems that will occur, if neither the given text nor the initial model is faultless.

Let an architecture documentation include the sentences:
"The common component consists of multiple parts. It includes the util, webapi and exception module."

The left side of Figure 4.2 is the model of the architecture. The right side is the goal model. Thus, the left initial model should be transferred into the right goal model. Compared to the textual description it stands out that some components of the initial model have other

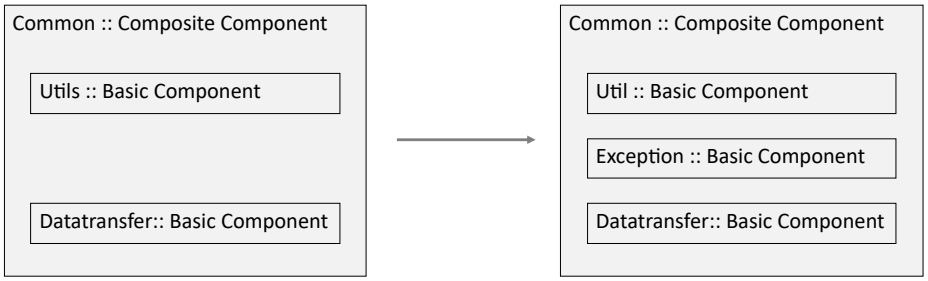


Figure 4.2.: The transformation of a model instance to a goal model instance. The model instance is missing an element

names, others are missing or additional to the described parts. Moreover, they have other types than described.

Thus, the target is to identify possible model elements in the text and connect the textual and model elements. The approach recognizes the underlined parts of the text as textual elements. They have a probability to be detected correctly. As mentioned in Chapter 1, the documentation as well as the model can include faults. Therefore, nothing can be connected with 100% certainty when searching for connections (without assumptions of correctness). Most likely connections, like connection of elements with the same name and type, or similar relations can be seen as trace links, as a (close to 100%) sure connection.

Model		**Text**	
Element Name	**Element Type**	**Element Name**	**Element Type**
Common	Composite Component	common	component
Utilis	Basic Component	util	module
-	-	exception	module
Datatransfer	Basic Component	-	-
-	-	webapi	module

Table 4.1.: Example 1: An example of trace links

In Table 4.1 the first two entries represent trace links. The described elements and the model elements are obviously the same. The question arises what should be done with the other possible elements that have no secure connection. For this reason hint links are used. As described above, hint links can connect two elements to be the equal (with a probability), or indicate whether a described element should be in the model.

Table 4.2 shows this. The already found trace links are hint links with a probability of 90% or higher. This threshold is arbitrary and can be adjusted dependent on the size and quality of the specific documentation. The next three entries indicate whether the mentioned element should be in the model. The probabilities for this metric could be calculated by the amount of references in the text or the correlations with elements already included by trace links. In this scenario they are just assumed and sampled. The last entries evaluate the connections between possibly similar elements between model and text. While the *webapi* itself is unlikely to be contained by the model it is more unlikely that it is similar to the *datatransfer component*. For the exception it is not that unlikely, as

it is described as a part of the *common component*, such as the *datatransfer component* in the model.

Model		Text		%
Element Name	**Element Type**	**Element Name**	**Element Type**	
Common	Composite Component	common	component	100
Utilis	Basic Component	util	module	90
-	-	exception	module	80
Datatransfer	Basic Component	-	-	90
-	-	webapi	module	50
Datatransfer	Basic Component	exception	module	50
Datatransfer	Basic Component	webapi	module	30

Table 4.2.: Example 1: The resulting hint links

After that evaluation, the trace and hint links should be proposed to the user. After checking them, the user is able to see possible faults (like elements proposed as hint links, but without a verified connection) and repair the documentation and model. Thus, the goal model of Figure 4.2 can be reached, while the documentation and the model can be corrected efficiently.

As described in the example, the use of hint links requires multiple thresholds. The upper threshold (*t-threshold*) defines when a hint link becomes a trace link. Thereby, the *t-threshold* defines the certainty of a hint link so that the user does not need to verify it. The lower threshold (*u-threshold*) defines when a hint link would be proposed to the user. Thus, the *u-threshold* is a lower bound for the correction of the documentation and model. If the *u-threshold* is too high, some found faults are not proposed to the user and thus not corrected. Otherwise, if the *u-threshold* is too low, every possible fault is proposed to the user and the approach would not be helpful. Therefore, it is indispensable to adapt the *u-threshold* to the examined documentation and model.

4.2. Detailed Approach

Section 4.1 gave an overview of the approach. In this section, the approach is presented and discussed in more detail. For this reason, the following enumeration sums up the processing order:

1. Extracting text information from architecture documentation
2. Extracting model information from the architecture model
3. Creation of hint links
4. Identification of trace links

Each step can cause various different challanges. In the first step, there is the problem of multiple sources: Often, architecture documentation includes images, graphs, or models. In this approach, these figures are excluded. In future work, it should be examined how this exclusion affects the results and how textual approaches could deal with figures.

The costs of formulating a graphic (like a UML sequence diagram) would be high, because it would have to be adapted to the writing style of the rest of the documentation. Beyond that, the description should not fall into a pattern. A pattern would be easy to use for the extraction of model elements, but not expressive for the approach. Another possibility would be to include the graphics as additional models to the approach. This can be done very easily, while the approach is not restricted to one model. However, in this thesis this is not done. If someone would use the approach for his (already existing) documentation, he would have to transfer every graphic into a model to get comparable results. By this he probably would get better results, because more information can be used. In this work, the focus is more directed on how to connect, and whether the approach can aid with corrections. Therefore, the results of the worst case (without any additional effort) are more interesting.

Natural language has very complex structure and needs many, different syntactic and semantic analyses, to be broken down for machines. Therefore, many related works use restricted or formal languages. The disadvantage of formal languages is that they have to be learned by the human writers of the input texts. The results are often stiff formulations, translatable into other models. In my opinion, the work with these structured languages is not expedient to approximate natural (free formulated) language. In this thesis only architecture documentations in natural language are used.

For preprocessing the textual input INDIRECT (and therefore also PARSE) is used. As described in Chapter 3, the PARSE based approach generates a graph of the input text and annotates it with different syntactical and semantical information. With this preprocessing, common used linguistic information is accessible for this approach.

4.2.1. Text Extraction

After preprocessing, the approach uses the linguistic information included in the PARSE graph to recognize and extract possible elements. In this work, this step is done with multiple analyzers. Every analyzer observes different types of information. For example: The POS analyzer is looking for part of speeches, while another one is looking for dependencies.

As mentioned in the introduction of this Chapter 4, elements can be every instance or relation from the model. Names and types of such elements do not always appear together in the text, as the example of Section 4.1 shows. Moreover, the example shows that the terms for the same element type can vary. For these reasons, the approach does not search for whole elements at the text extraction level. The approach searches for single names and types of elements. The analyzers of the text extraction level classify nouns as *names* and *types*. For all remaining nouns, the category *name or types* (*nort*) is created. Thus, this first level creates a base for the next levels to build upon (as can be seen in Figure 4.3). For further levels, the probability of each stored information is relevant. The probability in the text extraction corresponds to the certainty of the correct classification. An example for the text extraction is given in Section 4.2.1.1.

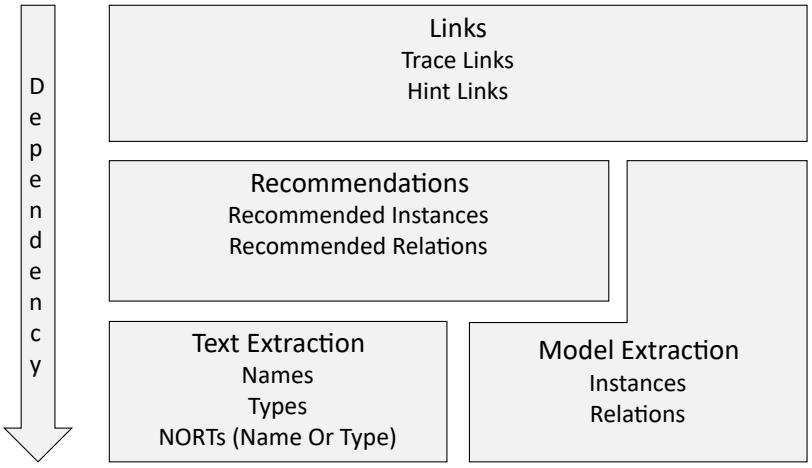


Figure 4.3.: The approach in levels: The text extraction and the model extraction are the base of the approach. The recommendations are built upon textual and model information. The creation of links depends on the information of recommendations and the model.

4.2.1.1. Example 2: Text Extraction

This example shows how the textual extraction works. In the first step of this approach, the text is searched for nouns. Then these nouns are classified as *names*, *types*, and *norts*. In this example multiple analyzers (simulated by different rules) run over the text and classify the nouns. After that, the classification results are stored with their occurrences, as well as their certainty for the classification result.

„The Common component containscommon utilities.
Package overview containscommon .exceptions, common.datatransfer.
common.exceptions containscustom exceptions.
common.datatransfer containsdata transfer objects.
common.datatrnsfer package containslightweight data transfer object classes for transferring data among components.
They can be combined in various ways to transferstructured data between components.“

Figure 4.4.: Example 2: The initial input text

Let Figure 4.4 be the input text of our approach, where all nouns are already marked by an analyzer. Adjectives that could be nouns or could be included by a term are dashed underlined. Then, the analyzers will have to classify these as names or types. For this example it is assumed that typos or different word forms are recognized as similar. Moreover, potential relations could be marked. For simplicity within this example only relations are recognized that are identified by a point-wise separation of two terms.

To simulate the analyzers, the following rules are applied to the text:

1. A noun is a type, if it occurs in plural.
2. A point-wise separated term represents an *separated* relation between both parts. Both parts are classified as names.
3. A noun is a name, if it is part of a relation and its not a type.
4. A noun is a name, if it occurs after a definite article.
5. A noun is a name, if it occurs before or after a type.
6. A noun is a type, if it occurs before or after a name.

Table 4.3 shows the results of the analyzers. The percentages for the classification results are based on the amount of rules and occurrences that confirm the classification result.

At the beginning of the text the words *common* and *component* can be classified with rules 1,2,4,5, and 6. Because of the classification of common as a name, the first two occurrings of *common* are added to the classification result. Additionally, the next two terms, *package* and *overview* this can be done. *Package* occurs as type after a point-wise separated term, which makes *overview* to a name. Therefore, they are classified as norts. Norts are nouns that can not be unambiguous classified as name or type and thus they can be used as both.

Term	**Occurrences**	**Name**	**Type**	**Nort**	**Rules**	**%**
common	Common, common, common.exceptions, common.datatransfer	x			2, 4, 5	90
component	component, components		x		1, 6	80
utilities	utilities		x		1, 6	60
package	Package, package		x		6	50
overview	overview	x			5	10
exceptions	common.exceptions, exceptions	x			2	50
datatransfer	common.datatransfer, data transfer, common.datatrnsfer	x			2, 5	70
objects	objects, object classes		x		1, 6	60
classes	classes		x		1, 6	10
data	data			x		-
ways	ways		x		1	10

Table 4.3.: Example 2: The results of the noun classification

With *exceptions* the problem gets more difficult. The problem is that *exceptions* occur as both: Type (because of the plural) and name (because of the point-wise separations). In this case, the name always outvotes the type. The reason for this is that the selection

for names is a lot more restricting than the classification of types. The next new term *datatransfer* occurs in a lot of different notations. Again, rule 2 classifies it as a name. According to rule 6 *objects* is a type. For *classes* no explicit rule is given. This noun follows a name and a type. *Objects* could be a shorter term for *object classes*. Thereby, the term *object classes* is added to the occurrences of *objects*. The term *classes* itself is classified as type (because of rule 1). For the term *data*, it is ambiguous if it is in singular or in plural. Therefore, the term is classified as a nort. The last term *ways* is classified as nort. Table 4.4 shows the results of the relation extraction. The entries of the *Term* columns are identifiers for the according rows in Table 4.3.

Type	**1. Term**	**2. Term**	**%**
separated	common	exceptions	70
in	common	datatransfer	70

Table 4.4.: Example 2: The extracted relations of the text

4.2.2. Model Extraction

The model extraction is (like the text extraction) on the first (bottom) level of Figure 4.3. The extracted information is classified as instances and relations. Instances are defined by a name and a type. Relations are constructs of at least two instances and a type that identifies the kind of the relation. In this approach, for the sake of simplicity, the directions of relations are not respected. An example for the model extraction is given in Section 4.2.2.1.

4.2.2.1. Example 3: Model Extraction

Let the intial model be given as in Figure 4.5.

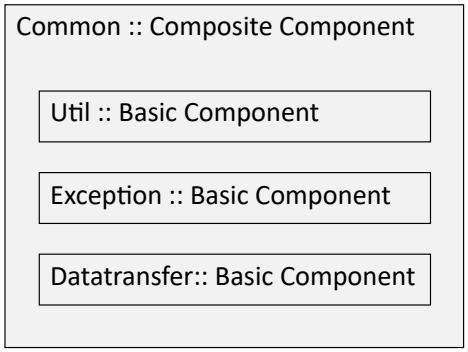


Figure 4.5.: Example 3: The initial input model

The initial model elements are transferred to an internal representations: Instances are stored as a name and a type (as in Table 4.5). Relations are represented by at least two instances and a type that describes the meaning of the relations (like in Table 4.6). In case of this model only one relation is included: *in* defines the containing of the first included instance in the rest of the included instances.

Name	Type
Common	Composite Component
Util	Basic Component
Exception	Basic Component
Datatransfer	Basic Component

Table 4.5.: Example 3: The extracted instances of the model

Type	Instances
in	Util, Common
in	Exception, Common
in	Datatransfer, Common

Table 4.6.: Example 3: The extracted relations of the model

4.2.3. Recommendation Generation

Based on the textual and model information recommendations for instances and relations can be created. This second level (mid layer of Figure 4.3) enables the approach to combine element parts of the text extraction and encapsulate them as instance. After the recommended instances are defined, the approach searches for recommended relations between them. All recommendations have probabilities. The probability of a recommendation corresponds to the probability of the existence of a recommendation with the specified name/nort and type/nort. This probability can be increased or decreased by analyzers using the extracted model information. If some recommended instances have the same type as some model instances the probability increases that they occur in the goal model. Another idea would be to increase the probability with the amount of occurrences in the text as measurement for relevance.

4.2.3.1. Example 4: Generating Recommendations

In this example the creation of recommendation is shown. For this the extraction results of the examples 2 and 3 are used (see Section 4.2.1.1 and Section 4.2.2.1). The names and types from the text extraction are compared to the names and types found in the model. If a name of an instance and a name are similar, their probability of being a name increases (in the classification results from the text). If a type or nort is found as a name, or a name or nort is found as a type of a model element its classification is changed. This leads to the updated text classification results in Table 4.7.

Term	Occurrences	Name	Type	Nort	%
common	Common, common, common.exceptions, common.datatransfer	x			100
component	component, components		x		100
utilities	utilities	x			80
package	Package, package		x		50
overview	overview	x			10
exceptions	common.exceptions, exceptions	x			70
datatransfer	common.datatransfer, data transfer, common.datatrnsfer	x			90
objects	objects, object classes		x		60
classes	classes		x		10
data	data			x	-
ways	ways		x		10

Table 4.7.: Example 4: The updated names, types and norts of the text extraction

For comparison additionally partial terms are used. Thus, component is recognized as a type of the model. In this example the *utilities* are identified as *Util*. Thus, the classification result changes the kind, but in contrast to *common* or *component* the probability is not increased to 100%. For relations this is done equally. In contrast to instances, the describing term of the relation is not crucial, but the involved instances.

Type	1. Term	2. Term	%
separated	common	exceptions	90
separated	common	datatransfer	90

Table 4.8.: Example 4: The updated relations of the text extraction

Like in the examples before, the analyzers for the creation of recommendations are simulated with rules:

1. If the text contains a name type, name nort, type nort or type name combination this connection is encapsulated in an instance of that name/nort and type/nort.
2. If a name is contained by the instance names of the model, it is added to the recommendations.
3. If a name has a probability higher than 50% it is added to the recommendations.

The application of these rules leads to Table 4.9. In this table the textual information is formatted into instances. Therefore, name and types from the results of the text extraction are combined. The entries of the columns *Name Term* and *Type Term* are used as identifier for the corresponding row in Table 4.7. The first rule leads to the first two results. The second one leads to the last three. Therefore, the deployment of the third rule is not needed. However, it represents the possibility that the probability of a name from the extraction level can be sufficient to create a recommendation of it. As usual, the probabilities of this example are roughly estimated. It represents the urgency of the element to appear in the goal model. For relations this is done vice versa (as can be seen in Table 4.10).

Name Term	**Type Term**	**%**
common	component	99
overview	package	30
datatransfer	objects	70
common		97
utilities		90
exceptions		80
datatransfer		95

Table 4.9.: Example 4: The recommended instances

Type	**1. Term**	**2. Term**	**%**
separated	common	exceptions	90
separated	common	datatransfer	90

Table 4.10.: Example 4: The recommended relations

4.2.4. Link Generation

Up to this point, hint links and trace links were always created in one step. As described in Section 4.1, hint links and trace links are treated as two different kind of links. Trace links were described as hint links, with a probability above a threshold (t-threshold). In further sections of this work they will be defined like this. Especially, hint links were defined as a single element, or a connection between an element found in both the model and the text. I split that hint link definition from above in two parts: First, a hint link with a single element is represented by recommendations. Recommendations specify a specific element (name and type combination) extracted from the text. Its probability is the probability of occurring in the goal model. Therefore, all recommendations with a probability over a specified threshold are hint links. The second part, the hint links between model and text are represented by instance and relation links. These are connections between extracted model instances/ relations and recommended instances/ relations. These connections have a probability for their connection. The probability measures the similarity of both, the model and the textual element. Whether the connection is treated as a hint or trace link depends on that probability.

4.2.4.1. Example 5: Creation Of Hint Links

For this example the knowledge of the examples Section 4.1.1, Section 4.2.1.1, and Section 4.2.2.1 is needed. In this last, and highest level of the approach the hint links are chosen. Therefore, similar to the recommendation creation it is searched for connections between the recommendations (textual information) and the extracted information from the model. Here, the task is to evaluate the similarity between two instances/ relations. For instances this is done, as usual, by the similarity of the identifying terms. For relations, it is done by the involved instances.

With the application of this rule in Table 4.11 and Table 4.12 could be the results:

Model		**Text**		**%**
Element Name	**Element Type**	**Element Name**	**Element Type**	
Common	Composite Component	common	component	99
Util	basic component	utilities		80
Exceptions	basic component	exceptions		90
Datatransfer	basic component	datatransfer	objects	85

Table 4.11.: Example 5: The instance links

Model		**Text**		**%**
Type	**Instances**	**Type**	**Instances**	
in	Exceptions, Common	separated	common. exceptions	99
in	Datatransfer, Common	separated	common, datatransfer	99

Table 4.12.: Example 5: The relation links

The results of this step include a probability, representing the matching of both instances/ relations. With these results and a threshold, hint links can be proposed to the user. If the threshold was set below 80% this would include all relation and instance links of this example. Moreover, for missing artifacts the recommendations can be proposed to the user. In this case, the suggestions would not change. If the user rejects a similarity, for example the equality of *datatransfer basic component* and *datatransfer objects*, another recommendation could slide up and be proposed. In this case, this would be just the name *datatransfer* as an equal element to *datatransfer basic component*. Thereby, *objects* would not be found as potential type of the model. If some model instances or relations are missed, like the *in: Utils, Common* relation, they can be proposed as failure-prone. Thus, the user is able to add more documentation to it or update the model.

As a result of this approach, all possible recommendations, hint links, and trace links are proposed to the user. In future work, he could verify or reject them. Thereby, new knowledge would be available and the process would have to evaluate its data again. Thus, other recommendations, hint links, and trace links based on the verified elements would have an increased probability, while rejected elements would have a decreased probability.

In the introduction of Section 4.1.1 the process was described as an approach without the assumption of correctness. The procedure might not seem to work like this. However, the approach works from the textual side against the model side. Nevertheless, the model is not seen as 100% correct. If no textual element would match to a model element, it can be seen as failure or (at least) as insufficiently described. Thus, inconsistencies can be found not only in the documentation, but also in models.

However, it is an open question how many inconsistencies can be contained so that the approach is still helpful. If most of the documentation and the model differ, it would be hard to find any connections between them. Thus, mostly all elements would be marked as possible failure prone. If only the model consists of mostly failures, the classification of recommendation based on model information could be faulty too. Thereby, the quality of the approach could decrease. If only the text would be very failure prone, this would lead to similar results. However, in future work, the efficiency of the approach, when model and text differs a lot has to be examined. Nevertheless, the approach will fail completely, if both, model and documentation consist of mostly failures. Thus, connections between model and text would be found, but would not be purposefully. Therefore, the precondition of this approach is that at least one of the inputs is claimed as mostly correct.

5. Architecture

This chapter describes the architecture of the approach. Section 5.1 gives an overview about the approach and its modular structure. After that, Section 5.2 provides more detailed information about the design of the specific parts.

5.1. High-level Architecture

The presented approach generates hint and trace links between an architecture and a documentation. This is done via multiple levels building upon each other. The four conceptual levels are already introduced in Chapter 4. Figure 5.1 shows the dependencies of each part by arrows pointing downwards to the part they are dependent of. In the implementation, each level (except the *Model Extractor*) is represented by an agent, holding a state. Thus, the agents are: *Text Extractor*, *Recommendation Generator*, and *Connection Generator*.

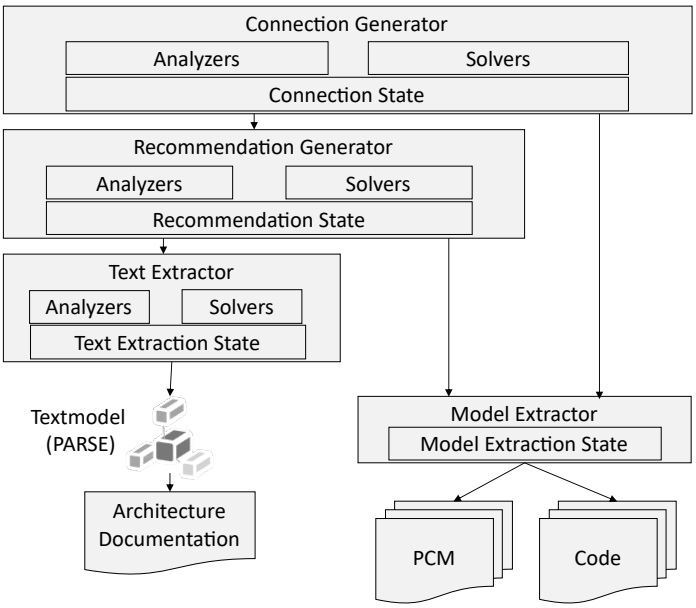


Figure 5.1.: High-level Architecture: The arrows represent the requirements of each level.

Agents inherit from the abstract PARSE agent. Therefore, the approach can be applied as a usual PARSE agent. Moreover, every agent has access to a PARSE graph. This is necessary for ensuring the possibility of a quick look up on the actual textual source.

PARSE agents are designed with the single responsibility principle. Therefore, each agent has is specific purpose. Furthermore, each PARSE agent has the opportunity to read and edit the PARSE graph. Usually, the graph is used as input and output of the PARSE agents. By inheriting from the PARSE agent, the possibility to edit the PARSE graph is kept in the implementation. In contrast to the usual agents, this feature is currently not used in this approach. The reason for this is the very specific purpose of this work. The goal of this approach leads away from the PARSE graph and its linguistic information. Thus, in this approach every agent has a state. Each state operates as a collection of knowledge for its agent. Especially for higher-level agents (like the connection generator) that do not need to run on graph this makes a big different. Instead of running through a large graph, they request a state downward to get a compact view on the current information. The information in the state of an agent is gained from analyzers and solvers of that agent. Each analyzer and solver can be seen as a specific search pattern. The modular design of them allows the anylzer/ solver to experiment different combinations to get the best results. Since analyzers are executed on a node of the PARSE graph, solvers only use the already extracted or generated knowledge of the states. In contrast to the agents, the *Model Extractor* is independent of the PARSE graph. Therefore, it does not implement the abstract agent from PARSE. Its goal is to preprocess models and bring them to a machine-readable form.

5.1.1. Analyzers And Solvers

As introduced, each agent (except of the *Model Extractor*) has analyzers and solvers. Analyzers work on PARSE graph nodes, whereas solvers only need the already gathered information in the states of the agents. For each level different analyzers and solvers inherit from abstract classes. In Figure 5.2 analyzers are divided in three subclasses: *Text Extraction Analyzer*, *Recommendation Analyzer*, and *Connection Analyzer*.

The analyzers of each agent have the possibility to use any information of the analyzers and solvers that ran before it. These references need to be initialized within the specific analyzer. Therefore, the subclasses offer a structure to specify all available resources for a class of analyzers (and solvers respectively). The *DependencyType* introduces a finer classification between the analyzers and solvers. The enum *DependencyType* specifies the actual dependencies of a concrete analyzer (or solver). Therefore, the enum encapsulates all combinations of possible requirements. This can be very helpful in further work, e.g. when parallelizing the analyzers. Moreover, it provides more structure and can help to understand all dependencies between multiple agents. The architecture of solvers is similar to the design of analyzers, as Figure 5.3 shows. Concrete instances of both can be created with a factory method, provided by every subclass. To improve the flexibility of the approach a configuration file specifies the analyzers and solvers that should be executed. Therefore, users can easily add, exchange, and leave out analyzers and solvers.

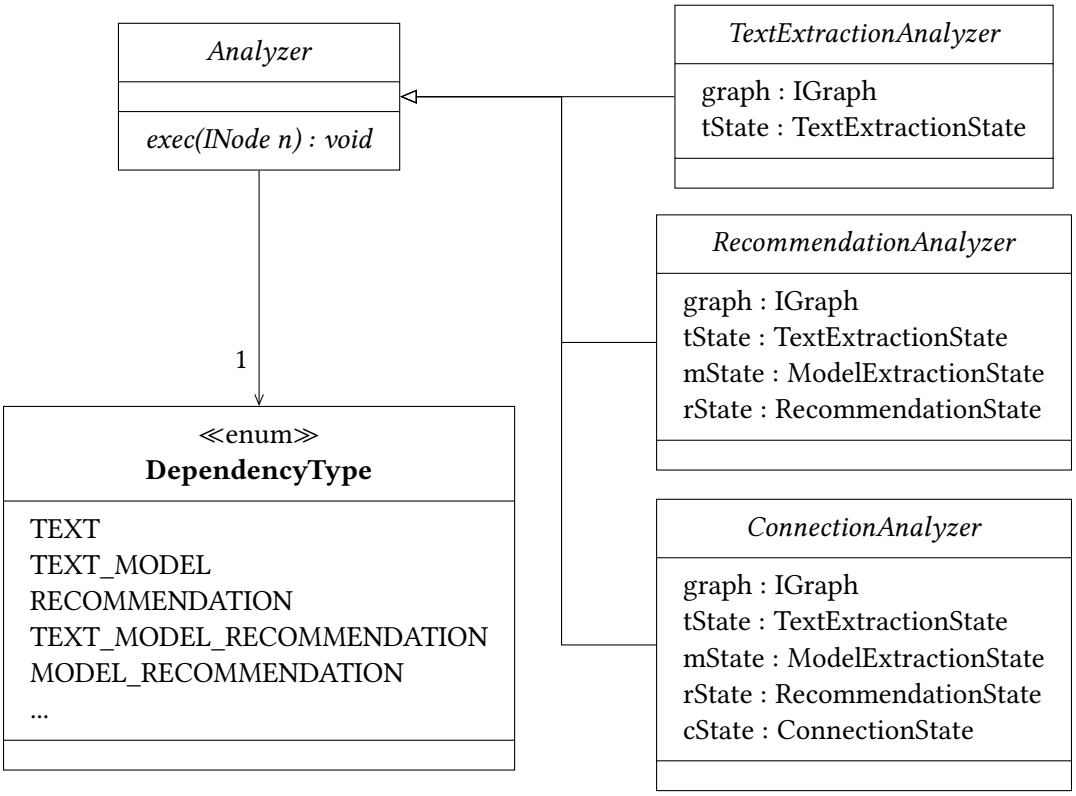


Figure 5.2.: Analyzer Architecture

5.2. Detailed Architecture

This section describes the architecture of the agents in detail. As described above, the approach consists of four core structures: The *Text Extractor* (Section 5.2.1) extracts textual information from the graph. The *Model Extractor* (Section 5.2.2) extracts instances and relations from the models. The *Recommendation Generator* (Section 5.2.3) generates suggestions for the goal model. The *Model Connector* (Section 5.2.4) creates the links between recommendation and model elements. As described in Section 5.1 every agent has exactly one state. Since the agents are PARSE agents and the analyzers and solvers are just editors of the states, the focus lies on the data stored in that states.

5.2.1. Text Extractor

The goal of the text extractor is to extract relevant information from the text. The text is contained by a PARSE graph. Relations can be extracted, too. The structure of the *Text Extraction* is represented by different classes.

As shown in Figure 5.4, extracted words are stored as *NounMappings*. A *NounMapping* consists of a list of PARSE *nodes*. The nodes link to the textual source and therefore to the occurrences. In a *NounMapping* the *occurrences* are all different appearances of the encapsulated word as it occurs in the input text. These occurrences are covered by a unified form: The *reference* of the mapping. The choice of the *reference* is important since

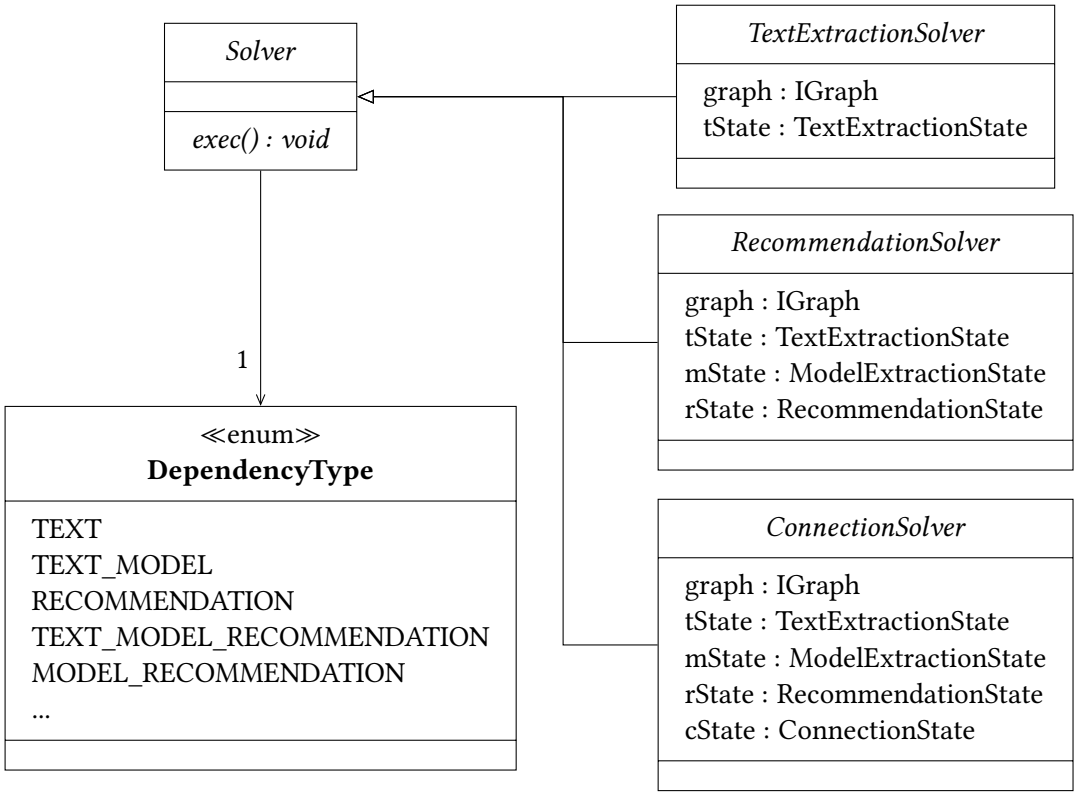


Figure 5.3.: Solver Architecture

it is used for the selection and association of new occurrences. Moreover, it is used for comparison in further levels. Therefore, it serves the further processes like a key of a dictionary to find and access *NounMapping*s. Each *NounMapping* has a *MappingKind*. This *MappingKind* identifies the type or the mapping. Thus, the mappings can be defined as *NAME*, *TYPE*, or the possibility of both *NAME_OR_TYPE* (short nort). The *probability* of a mapping represents the probability of that mapping to have the given *MappingKind*. The *MappingKind* can be changed. If the *MappingKind* is changed, the probability should be changed, too.

An example of a *NounMapping* would be the representation of the terms *car* and *cars*: An analyzer identifies both as different forms of the base term *car*. Moreover, the analyzer classifies *car* as a name. The analyzer transmits this information to the state. The state creates a *NounMapping*. It sets the *reference* of the *NounMapping* to the transmitted base term (*car*). Then the state adds *car* and *cars* to the *occurrences* and the related PARSE nodes to the *nodes* field. Since the analyzer classified *car* as *name*, the *MappingKind* of the *NounMapping* is set to *NAME*. Finally, the state sets the *probability* of the *NounMapping* to the certainty transmitted from the classifier.

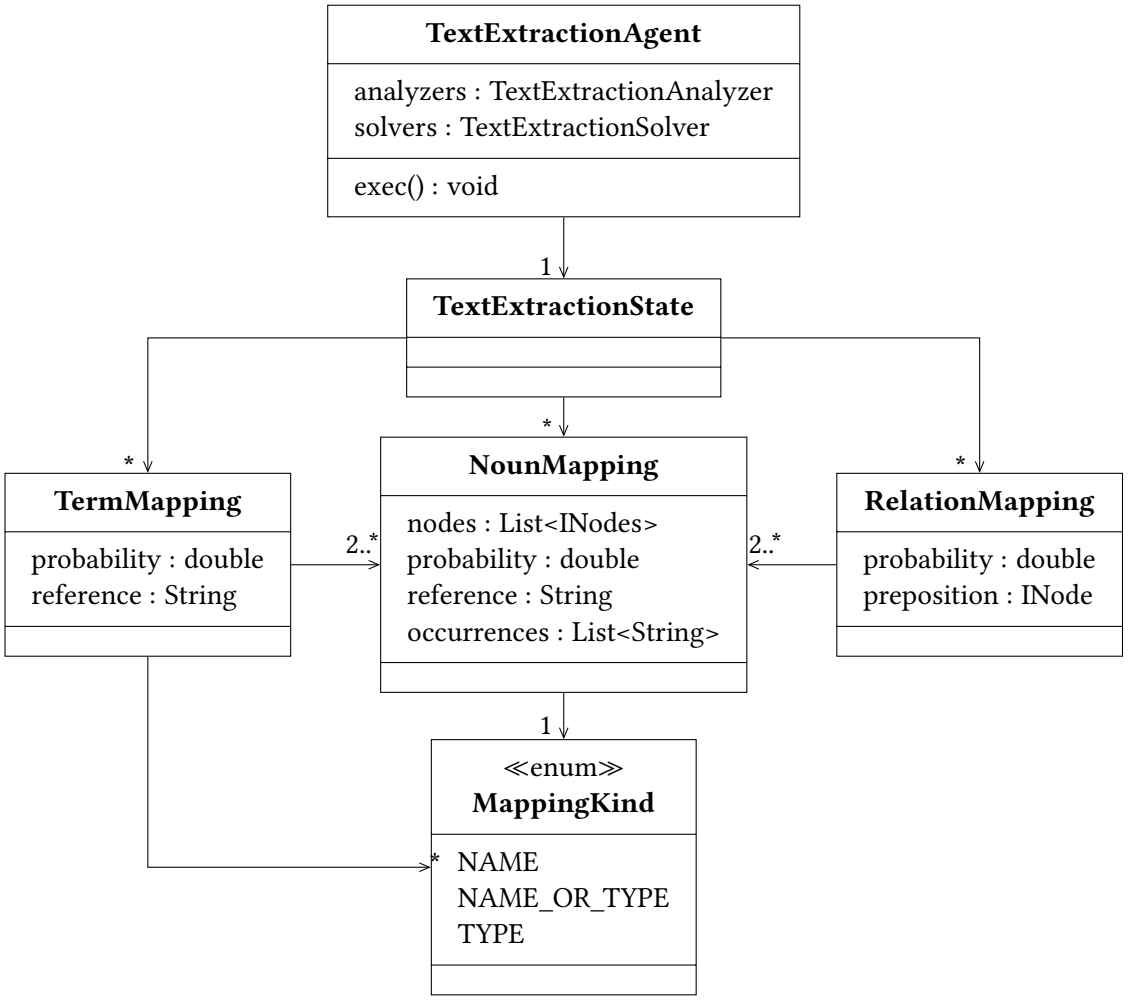


Figure 5.4.: Text Extractor Architecture

Textually identified relations are represented by the *RelationMapping*. A *RelationMapping* needs at least two *NounMapping*s. Additionally, it can have a *preposition*, given as a PARSE node. This *preposition* could be helpful, when working with directed relations. Up to this point, the approach considers only non directional relations. The *probability* of a *RelationMapping* is the probability that the *NounMapping*s have a connection in this constellation.

For terms that consists of more than one word (e.g. *car wheel*), *TermMapping*s are introduced. *TermMapping*s consists of at least two *NounMapping*s and a reference, to represent that term. Terms have a *MappingKind*. The *MappingKind* of each part (*NounMapping*) of the term should not counter the *MappingKind* of the *TermMapping*. Thereby, a name term could contain a name or nort, but no type mapping.

5.2.2. Model Extractor

The *ModelExtractor* is an interface for different model extractors. Therefore, different types of models should be representable by the *ModelExtractionState*. Because the approach works on text, only the terms of the names and types of model elements are important. The structure should enable backtracking to allow accurate links between textual elements and specific model elements.

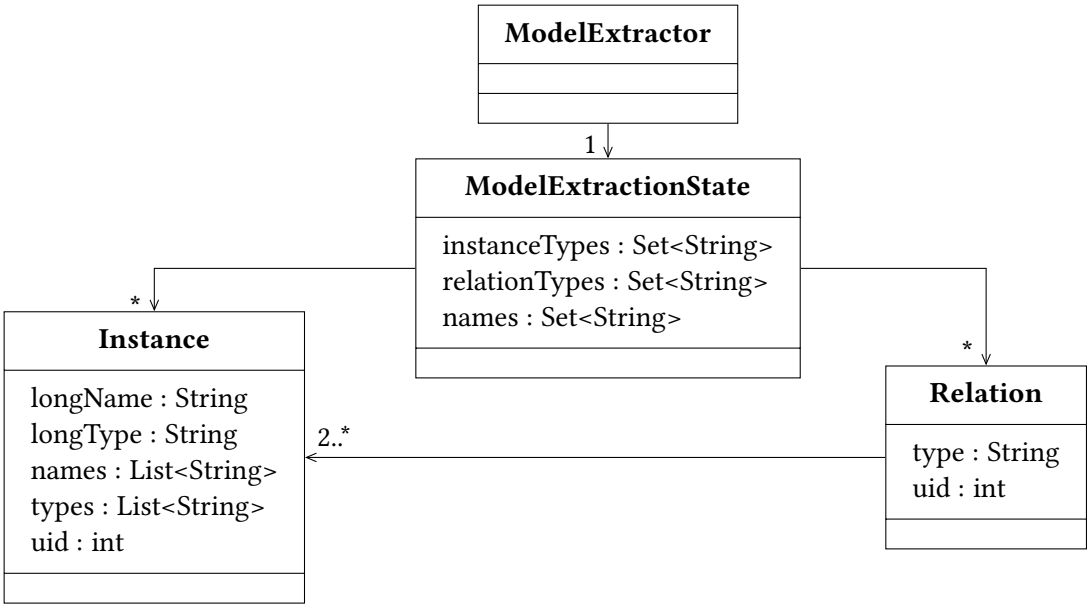


Figure 5.5.: Model Extractor Architecture

As can be seen in Figure 5.5, the *ModelExtractionState* consists of *Instance*s and *Relation*s. For better performance during the matching in further steps, their types and names are collected as sets and stored separately. An *Instance* is able to represent every model element, except for relations. It has a *longName* and a *longType*. These contain the fully classified name of the instances name or type. The longest name should be unique in the model. All parts of the name are stored in *names* (analogous for *types*). This enables to search for connections to textual elements by partial mentions.

For example: The given model is a UML class diagram that contains the class *car wheel*. This class is represented by an *instance*. The *longName* of the instance is the term *car wheel*. As *longType* the term *class* has been inserted. The *names* consist of the parts of the *longName*. Thereby, the terms *car* and *wheel* are written to the *names* field. The content of the *types* attribute is equal to the *longType*, since the long version of the type only consists of one term. If the model would have a type term of multiple words, like *composite component* (in PCM), the *types* were filled with the parts of the *longType*.

Like in the *RelationMapping*s of Section 5.2.1, each represented *Relation* has at least two *Instances* as participants. Like the *Instance*, a relation is linked to the specific model element by a *uid*. Moreover, a *Relation* has a *type*. This *type* defines the kind and meaning of the relation. An example for that would be “in” for a relation between a contained element and its container.

5.2.3. Recommendation Generator

The *RecommendationAgent* generates recommendations and suggestions of instances and relations to be contained by the goal model. Like every agent, it consists of multiple *analyzers* and *solvers*. The results of them are stored in the *RecommendationState*.

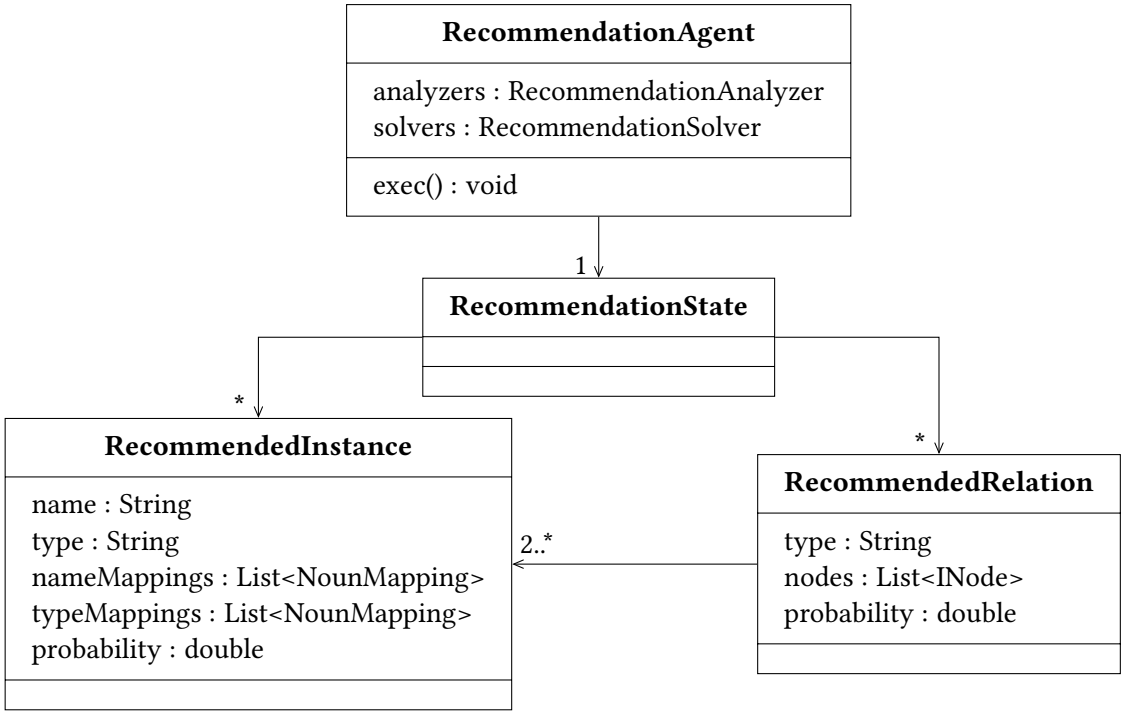


Figure 5.6.: Recommendation Generator Architecture

As shown in Figure 5.6, the *RecommendationState* consists of *RecommendedInstance*s and *RecommendedRelation*s. A *RecommendedInstance* has a *name* that works like the reference at the *NounMapping*s of Section 5.2.1; It can be used as the unified identifier of the encapsulated *nameMappings*. These *nameMappings* should contain only one element, but there are two exceptions. On the one hand, the name could be a term and therefore consist of multiple *NounMapping*s. On the other hand, multiple *NounMapping*s could be the same recommended instance. For the *type* of the *RecommendedInstance* this is built analogously. Moreover, a *RecommendedInstance* has a probability that measures the likelihood of the appearance in the goal model.

In Section 5.2.1 it was described how the *NounMapping* of the term *car* could look like. The same could have happened with *class*. In contrast to *car*, an analyzer classified the term *class* as a type. On the recommendation level an analyzer finds a textual connection between *car* and *class*. The analyzer classifies this connection as a recommendation. Thereby, a *RecommendedInstance* is created. The *name*of the *recommended instance* is given by the *name* of the name-mapping (*car*). As *type* the analyzer enters the *name* of the type-mapping (*class*). The mappings theirselfs are added to their respective collection (*nameMappings* and *typeMappings*). The probability of the *recommended instance* is set to the certainty transmitted from the analyzer.

Analogue to the other representations of relations in this approach, the *RecommendedRelation*s consist of at least two instance representations of its level (*RecommendedInstances*). For representing the semantic of the relation, a *type* can be stored. For referencing the *type*, PARSE nodes can be referenced. Similar to *RecommendedInstance*s, the probability represents the likelihood of this relation in the goal model. Currently this probability is only used for the occurrence of a connection between all participating instances. The semantic meaning of it can be used for direct proposings to the user or developed in future works.

5.2.4. Connection Generator

The *ConnectionAgent* represents the last step of the approach. Its analyzers and solvers create links between model and textual elements. Since the *RecommendationGenerator* creates recommendations for the goal model, its suggestions are used as representation of the textual information.

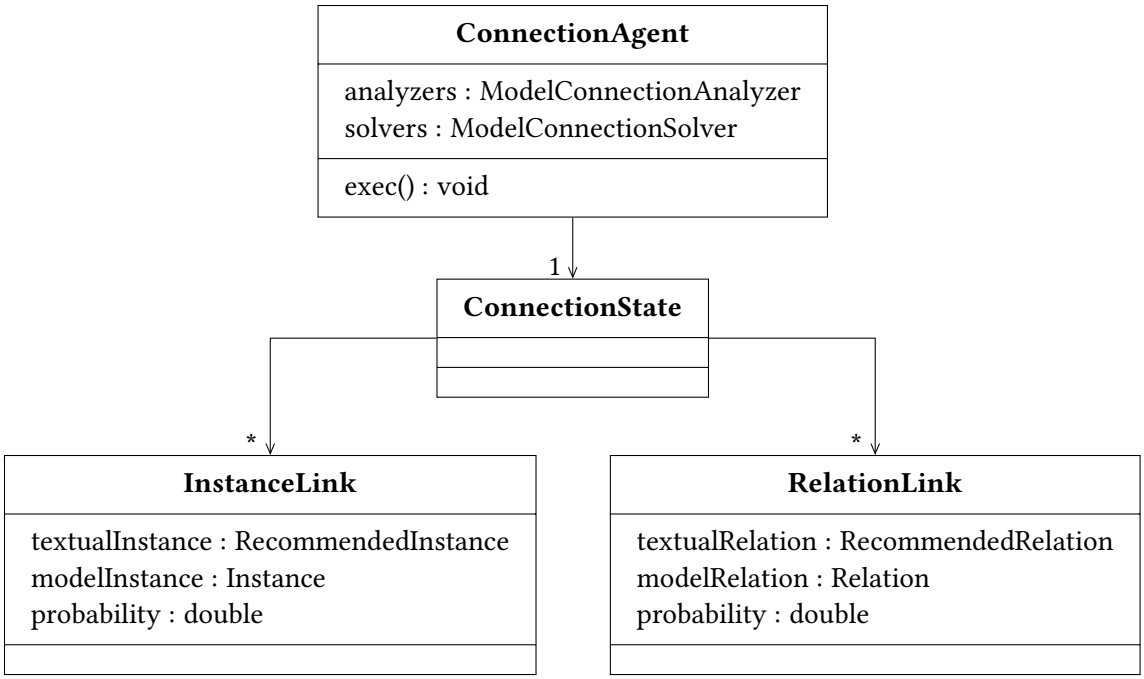


Figure 5.7.: Connection Generator Architecture

Figure 5.7 shows the structure of the *ConnectionState*. The *ConnectionState* consists of *InstanceLink*s and *RelationLink*s. *InstanceLink*s consist of a *textualInstance* represented by a *RecommendedInstance*, and a *modelInstance* represented by an *Instance* from the model. The probability of it measures the similarity of both instances. Two instances are similar if the names as well the types are similar. Chapter 6 will introduce the term similarity in more detail. If the approach had found the *RecommendedInstance car* of type *class* and an *instance* with *car* as *longName* and *class* as *longType* would exist, they would be connected within a *InstanceLink*. The *RelationLink*s consists of a *textualRelation* represented by a *RecommendedRelation* of the previous step, and a *modelRelation* represented by a *Relation* of the model. The probability of it measures the similarity of both relations.

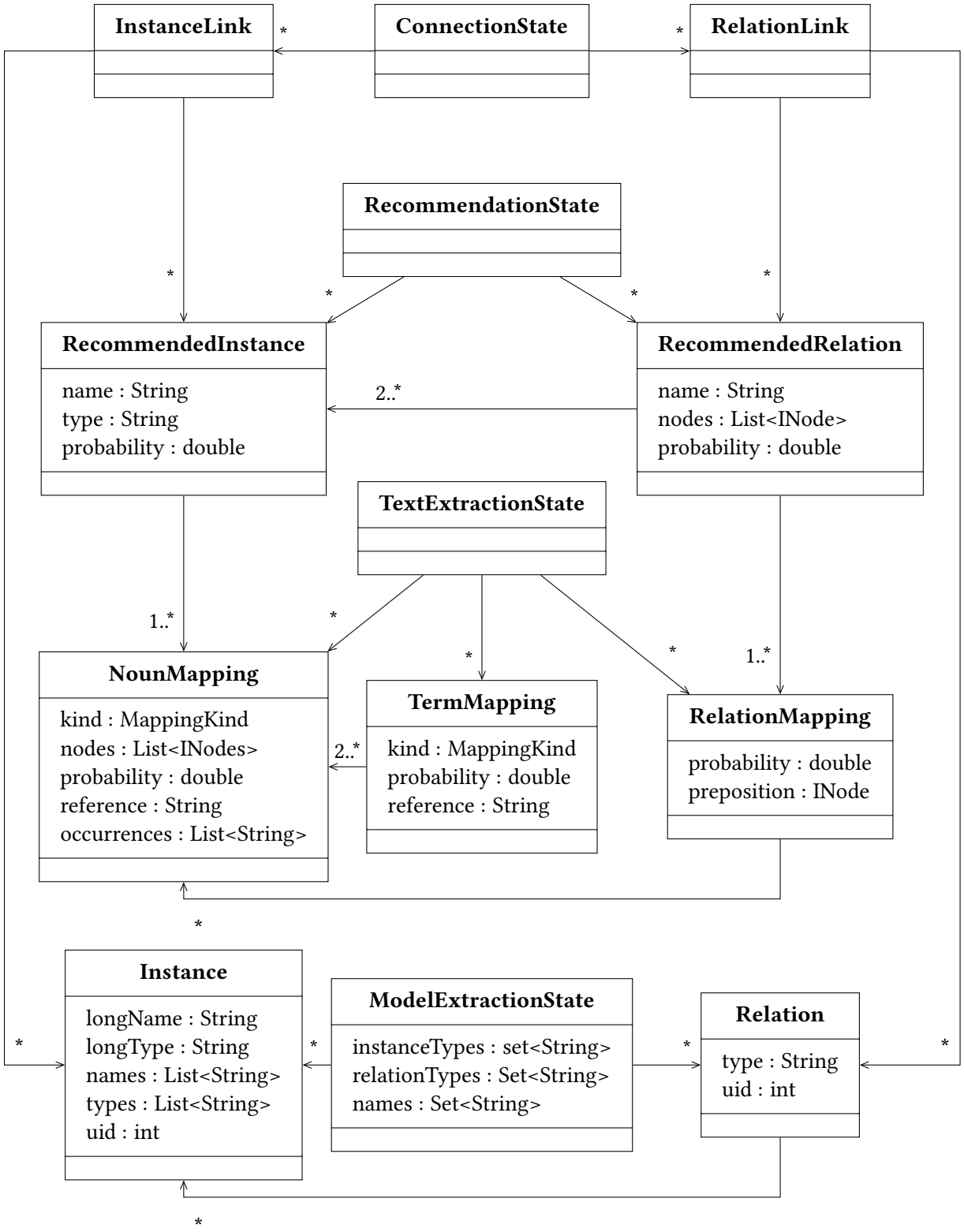


Figure 5.8.: Architectural relationships between all states

6. Implementation

In this chapter the implementation of the approach is presented. Therefore, the four parts of this approach (introduced in Chapter 5) will be presented in more detail, as well as the processes between them. Following from bottom to top, Section 6.1 describes the processes of the text extractor. Since only the structure of the model extractor is implemented yet, it will be skipped in this chapter. Then, Section 6.2 presents the logic behind the creation of recommendations. Section 6.3 gives an overview how links are created and evaluated. At last, Section 6.4 will describe the configuration of the process.

6.1. Text Extractor

The *TextExtractionAgent* is the most underlying agent in this approach. Its goal is to find and extract usable information of the text. This information includes names, types, other nouns, terms, as well as relations.

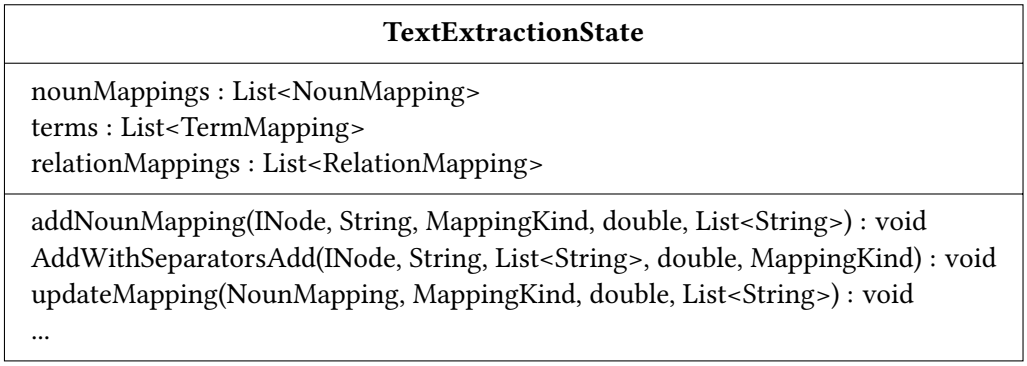


Figure 6.1.: Overview of the Text Extraction State as UML class

As shown in Figure 6.1, the state contains *NounMapping*s, *TermMapping*s, and *RelationMapping*s. As already described in Section 5.2.1, *NounMapping*s provide the structure to encapsulate multiple words of the same meaning. Thus, further processes do not have to run through the whole input graph, but can use the information provided by the *TextExtractionState*. *TermMappings* define a term of multiple words (for example *car wheel*). The included words are represented by *NounMapping*s. Both, *NounMapping*s and *TermMapping*s, are specified by the enum *MappingKind*. It classifies the mapping as name, type, or nort. Norts (name_or_type) is the class for not closer classified nouns. They can potentially used as both: name or type. *RelationMapping*s store relations. A *RelationMapping* consists of at least two *NounMapping*s and can have a preposition, linked to a PARSE node.

The state is edited by analyzers and solvers. To ease the interaction with the state, the analyzers or solvers request the adding of their potentially found mapping. The logic for the comparison to the existing mappings is done by the state itself. The next sections show how *NounMapping*s are built and how their attributes are used.

6.1.1. The Adding Of Basic NounMappings

In Section 5.2.1, *NounMapping*s were explained as a representation of words (actually nouns), mapped to their textual source. Figure 6.2 gives a short overview of the attributes stored by these mappings. To understand the next functions, it is necessary to understand the meanings of each attribute.

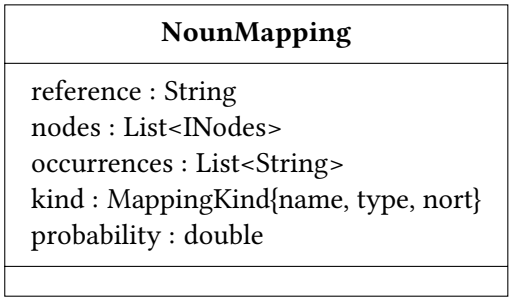


Figure 6.2.: NounMapping as UML class

The *nodes* are the PARSE nodes that represent the reference to the textual source. They contain the position, the actually token (word), and linguistic information. The words, as they appear in the nodes (or some equal modifications) are stored in the *occurrences*. Modifications can be done to ease the processing (for example by replacing separators), but should never change the similarity or the overall look of the word. The occurrences are unified in one string: the *reference*. The reference is important as it is used for the comparison between multiple *NounMapping*s. The textual similarity between a new word and the reference decides, whether both are similar and the word should be added to the existing mapping or not. Since the approach is based on a modular design, analyzers and solvers are creating noun mappings, and thereby determine these references. In this work, as reference often the occurrence itself is used. In the future it can be tested, if the results could be improved by using a base form or normalization of the occurrence. Moreover, each *NounMapping* has a *MappingKind*. This enum defines, the kind of the mapping and thereby the classification. The kind separates nouns in names, types or norts. It is important to classify the *NounMapping*s as correct as possible, since further steps are built upon this information. As described, the state controls every interaction with *NounMapping*s, while the analyzers and solvers just invoke the adding of mappings with a kind. Thus, the *TextExtractionState* has to weigh up between different statements of different analyzers if the kinds differ. For this evaluation the *probability* is important, since it contains the information of how certain the analyzers or solvers are that the mapping is of the given kind.

Algorithm 1 The add functionality of NounMappings to the Text Extraction State

```
function addNounMapping(node, reference, kind, probability, occurrences)

    separators ← set by configuration
    if reference contains any separators then
        partialRefs ← split(reference, separators)
        for all partialRef : partialRefs do
            AddWithSeparators(node, partialRef, probability, kind)
        end for
    end if

    nounMappingsWithNode ← all existing nounMappings with node
    nounMappingsWithRef ← all nounMappings with most similar references

    if nounMappingsWithNode is empty && nounMappingsWithRef is empty then
        createdMapping ← create new mapping with the parameters

    else if nounMappingsWithNode is empty then
        if #nounMappingsWithRef == 1 then
            currentMapping ← nounMappingsWithRef[0]
            addNodeToMapping(currentMapping, node)
            updateMapping(currentMapping, kind, probability, occurrences)
        end if

    else
        for all currentMapping : nounMappingsWithNode do
            updateMapping(currentMapping, kind, probability, occurrences)
        end for
    end if
end function
```

For the adding of a *NounMapping*, the state offers various interfaces. All requests of them are forwarded to the private method *addNounMapping(...)*, presented in Algorithm 1. The method *addNounMapping* needs multiple parameters: The *node* is the PARSE node that links to the textual resource and serves as a base for the new mapping. The *reference* is a word that unifies all words in this noun mapping and is similar to them. The *kind* is the kind of the future mapping. The likelihood for this kind is given by the *probability*. The *occurrences* contain all word forms, how they occurred in the text.

At first, the method checks, if the given *reference* contains any *separators*. Separators are characters defined in the configuration file. Since INDIRECT does not split at every separator like e.g. punctuations, colons, or spaces, it is not guaranteed, that the content is just a single word. . However, the approach is based on single words that are later combined and compared. Therefore, a reference should only consist of a single word. In this first part of the method, references with more than one word are handled. The reference is split along the separators into single words. For every single word in the reference, the method *AddWithSeparators* is invoked. This method will be explained in Section 6.1.2 later.

After that, all existing *NounMapping*s with the given node are collected. These mappings represent all mappings that refer to the same textual brick. Moreover, all existing *NounMapping*s with most similar references (compared to the given one) are collected. The term of similarity will be discussed later in Section 6.1.3. Thereby the collection contains all mappings that could represent the mapping-to-add. In case of equal similarity distances, *nounMappingsWithRef* can contain more than one result. The next steps of the algorithm depend on the entries of these collections.

If neither mappings with the node nor mappings with a similar reference have been found, the mapping-to-add is created. Thereby, it is added to the existing mappings of the state. If no mapping with the node can be found, it is searched for a mapping with a similar reference. The mapping with the most similar reference is taken and the given parameters (like occurrences and node) are added to the found mapping. Then, the method *updateMapping(...)* is invoked.

The *updateMapping(...)* method enables the updating of probabilities and the change of *MappingKind*s. The behavior of updating a mapping depends on the kind and probability of the stored mapping, as well as the kind and probability of the same mapping that is supposed to be added. A simple rule followed in this method is to specify only and not to generalize. This means that every mapping of kind nort can be changed to types or names, but no name or type can be set back to nort. This behavior is important to note when adding more analyzers or solvers, since they classify the instances. If a type should be updated to a name or vice versa, the probability for the "new" mapping has to be higher than the already stored one. For future work, a more granular probability consideration could be helpful. If the already existing mapping is updated, the *MappingKind* is changed and the probability is recalculated. In any case, the occurrences of the mapping-to-add are added and the probability is updated. At last, if mappings containing the node have been found, all of them are updated. Thereby, the occurrences are extended and the kind of the existing mapping is evaluated with the new statement.

In contrast to the previous case, all mappings are updated this time. If in the previous case more than one mapping with a most similar reference are found, the matching mapping can not be identified unambiguously. By doing nothing, the implementation improves the precision of the text extraction, but decreases its recall. It is an open question for the future work, whether it is more recommended to extend mappings only when they are assigned clearly, or simply all. On the other hand, if the node is already contained, the edit of the state is related to exactly all mappings that contain this node (as it represents the origin). Therefore, if multiple mappings contain the node and no separator is contained in the reference, the new information refers to all mappings that node is in.

6.1.2. The Adding Of NounMappings Containing Separators

In this section the method *AddWithSeparators(...)*, shown in Algorithm 2 is described. This method is invoked by *addNounMapping(...)*, described in Section 6.1.1. It is called if the reference of the new mapping contains a seperator. For every part of the reference split at the separators, the *AddWithSeparators(...)* method is invoked. Since nodes can be contained by multiple mappings, and references are not provably reliable for comparison, the addition of mappings with multiple references per one node has to be differentiated from the common one.

The *AddWithSeparators(...)* provides the functionality for adding mappings that should be added more insisting than on the common way. At first, the probability for adding a mapping with this function is calculated. This is done because this method works a bit different that the common one and is therefore more or less reliable (depends on the calculation function). Currently, the probability is decreased because the method provides a functionality to skip the adding function with the higher requirements. The percentage of the given probability that should be set as *AddWithSeparatorsProbability* can be set in a configuration file. After that, as in Section 6.1.1 all mappings with the node and those with the most similar references are collected.

If no existing mapping with the node can be found the method *addOrUpdate(...)* will be invoked. This method checks, if the given list of mappings that contain the reference (*mappings*) is empty. If this is the case, a mapping is created. After that, the approach searches for mappings with similar occurrences. The created mapping is then extended by the similar occurrences and their nodes. This step is done to adopt already classified occurrences to the possibly correct mapping. The approach is conservative and does not remove already existing mappings. Alternative: If the occurrences (per source) only should appear on time, they would have to be newly classified. This would lead to laborious circumstances and further classifications that have to run newly, if some data is already based on the mapping. In this thesis, I decided to focus on the results of the total possibilities, to secure at first that all mappings are found that should be found. If *mappings* contains mappings, *addMappingDependentFromReference(...)* adds the node to them and updates their kind with the given probability.

If no mapping with the node but at least one with a reference have been found the node is added to all of them. After that, the existing mappings are extended by the new occurrences and updated by kind and probability. The last case of this method is reached, if the node is already contained by a mapping. In this case, the method *addMappingDependentFromReference(...)* is invoked.

6.1.3. Similarity

As described in Section 6.1.1 and Section 6.1.2 the implementation is dependent from the definition of similarity. Especially, when extracting textual information and synchronize them to already stored information the similarity definition between two words (strings) is important. To vary between different graduations of similarity multiple thresholds can be set in a configuration file. These thresholds are used in the referenced methods. The methods for similarity are encapsulated in a static helper class, called *SimilarityUtils*. This class offers multiple ways for comparison. For the comparison of textual elements (and thereby, for the compare of two references) the Levenshtein distance and the longest common substring are used. It is important to notice that the comparison is not symmetric, since the Levenshtein distance is not.

The Levenshtein distance is an edit distance between two strings. If two characters of the same position are different, the distance is increased by one. This is done equally for insertions or deletions. The longest common substring and the Levenshtein distance are measured and evaluated separately. Upper and lower cases are ignored by the implementation. The accept of faults increases with the length of the original word.

Currently the metrics for similarity are relatively simple. There are much more metrics that can be used for word similarity (e.g. semantic metrics). By its modular design, functionalities and metrics can be exchanged or added easily.

6.1.4. Adding Term Mappings And Relation Mappings

The adding of term mappings and relations mappings is not that complicated compared to the adding of noun mappings. As described in the overview of this section, terms are connections of multiple noun mappings that represent a term of multiple words (like *car wheel* or *composite component*). Terms have a mapping kind that declares whether the term is a name or a type. Currently, the approach does not create terms of kind nort, even if it is possible. If an analyzer or solver adds a term to the state, it is secured that it is not yet included. To check that, the state searches for already stored terms that consists of the same mappings and have the same mapping kind. If some are found, their probability is updated. Otherwise, a new term is created and added to the state. Note that by this process mappings with the same content and different kinds can exist simultaneously.

The adding of relations currently only consist of a comparison of the lists of participating noun mappings. The order of them is considered. If does not occur in the stored relation mappings of the state it is created and added to it. Otherwise, it is discarded.

6.1.5. Analyzers And Solvers

All mappings are created by analyzers or solvers of the *TextExtractionAgent*. As explained in Section 5.1, analyzers run through the graph, while solvers only use the already stored and from their level available resources. All analyzers/ solvers of this agent are *TextExtractionAnalyzer/ Solver*s.

Algorithm 2 The add functionality Of NounMappings that contain a separator in their reference

```
function AddWithSeparators(node, reference, probability, kind, occurrences)

    AddWithSeparatorsProbability ← calculateProbability(probability)
    mappingsWithNode ← all existing nounMappings with node
    mappingsWithRef ← existing nounMappings with the most similar reference

    if mappingsWithNode is empty then
        addMappigDependentFromReference(mappingsWithRef, ... )

    else
        for all currentMapping : mappingsWithNode do
            if any occurrence contains any separators &&
               currentMapping.reference not similar to reference then
                addMappigDependentFromReference(mappingsWithRef, ... )

            else
                addOrUpdate(mappingsWithRef, ..., AddWithSeparatorsProbability, ... )
            end if
        end for
    end if
end function


function addOrUpdate(mappings, node, reference, probability, kind, occurrences)

    if mappings is empty then
        createdMapping ← create new mapping with the parameters
        mappingsWithOccs ← existing nounMappings with similar occurrences
        extendWithOccurrences(mappingsWithOccs, reference, createdMapping)

    else
        for all mapping : mappings do
            addNode(nounMapping, node)
            updateMapping(mapping, kind, node, probability, occurrences)
        end for
    end if
end function
```

The *TextExtractionAgent* has the following analyzers and solvers:

1. Noun Analyzer
2. Incoming Dependency Arcs Analyzer
3. Outgoing Dependency Arcs Analyzer
4. Separated Names Analyzer
5. Article-Type-Name Analyzer
6. Multiple Part Solver

Since the state manages all mappings, the analyzers and solvers just add their new information to it. How the information of the analyzers and solvers is used depends on the behavior of the specific state (in this case the text extraction state).

Noun Analyzer The goal of the *Noun Analyzer* is to find and extract the nouns. For this, it uses the POS (Part Of Speech) tags of the PARSE graph. PARSE uses the nltk tagset [23] with four variants for nouns: *NN* stands for common nouns, *NNP* stands for proper nouns, *NNS*, and *NNPS* are their plurals. If the current node is a common noun in plural, it is classified as type. Otherwise, it is classified as nort, and can be classified more specifically by other analyzers and solvers.

Incoming Dependency Arcs Analyzer The *Incoming Dependency Arcs Analyzer* checks the incoming dependency edges of the PARSE graph, created with the Dependency Parser agent of INDIRECT. Figure 6.3 shows an example, how the results of such a dependency parser could look like [1]. Table 6.1 gives an overview of the used dependency tags and explains them briefly. If the tags occurs at an incoming dependency edge of the current node n, the additional requirements are checked. If they are satisfied a noun mapping with the node and the concluded type is created. If the additional requirements are not fulfilled, a nort is created. The idea behind that additional requirements is the following: For a secure classification as type, the previous node is usually required to be an indirect determiner. Thereby, the noun loses its uniqueness. Thus, it can be seen more as a type of an object than a unique, specific object. For more information about the tags have a look on [20] or [30].

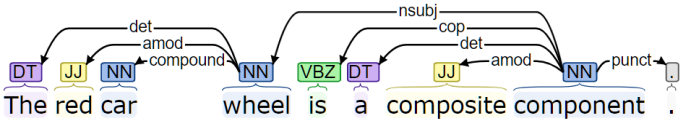


Figure 6.3.: The analysis of a sentence with a dependency parser.

Tag	Meaning	Additional Requirements	Conclusion
appos	n is a modified noun phrase		nort
dobj	n is a direct object of a verb (accusative)	previous node is indirect determiner	type nort
iobj	n is an indirect object of a verb (dative)	previous node is indirect determiner	type nort
nmod	n is modified by a noun or noun phrase	previous node is indirect determiner	type nort
nsubj	n is the syntactic subject of the phrase		nort
nsubjpass	n is the syntactic subject of a passive clause	previous node is indirect determiner	type nort
pobj	n is the object of a preposition	previous node is indirect determiner	type nort
poss	n is the head of a noun phrase of a possessive relationship		name

Table 6.1.: The mapping of incoming dependency tags to concluded noun mapping kinds

Outgoing Dependency Arcs Analyzer This analyzer checks the outgoing dependency edges of a node. For this, the edges have to be created first by the *Dependency Parser* of INDIRECT. In contrast to incoming dependency edges, there are less dependency tags that can be used for identifying and specifying nouns. Table 6.2 shows the used tags and explains them for a current node n. If a current node has an outgoing dependency edge of that type, a noun mapping of the concluded mapping kind is created. The reason for the conclusion as type, in case of *num* and *predet* is: If a noun has a predeterminer or is enumerable, it can not be unique. Therefore, it can be seen as type of something and is no unique identifier for an instance. For more information about the dependency tag set see [20] or [30].

Tag	Meaning	Conclusion
agent	n is the agent of a passive verb	nort
num	n is target of a numeric modifier	type
predet	n has a predeterminer (for example “all”)	type
rcmod	n is modified by a relative clause	nort

Table 6.2.: The mapping of outgoing dependency tags to concluded noun mapping kinds

Separated Names Analyzer The separated names analyzer identifies nodes containing separators. Then it adds it as noun mapping of kind name to the extraction state. In Section 6.1.2 the node value (word) is split at the separators. Examples for this would be: *intelligence::ai* or *car.wheel*. Then, the method adds each part is separately to the text extraction state. The separators for that process are stored in a configuration file. For this thesis, they contain ., ::, and :. Note that INDIRECT (see Section 3.2) should be used, such that usual punctuation marks can be handled.

Article-Type-Name Analyzer This analyzer checks the current node and its surrounding for two specified patterns. In both cases the current node has to be previously classified as a nort (or more specific kind). After the classification, a noun mapping of the concluded mapping kind is created.

Article Type Name:
: If the previous node of the input is a type and the predecessor of that one is an article, the current node is classified as a name. For example “The component a is ...”

Article Name Type:
: If the previous node is a name and the predecessor of that is an article, the current node is classified as a type. For example “The ai component is ...”

Multiple Part Solver Currently, the multiple part solver is the only solver of the text extractor. It identifies terms in the current extraction state. At first, it runs through all noun mappings that are names and checks if the previous nodes are also nouns. If the previous node is a nort or a name, the solver creates a term mapping of both nodes. When searching for type terms this is done analogously: if the previous node of the type declared node is a nort or type, a term mapping of both nodes is created. The term mapping always gets the most specific mapping kind (type/ name). Term mappings of combinations between name and type are not admitted. Since the text extractor just identifies names and type, combined later in the recommendation generator, this would disagree the design.

6.2. Recommendation Generator

The recommendation generator was introduced as the first instance that connects types and names found by the text extractor. Furthermore, it is the first layer that is able to use information from the model, although its goal is not the connection between text and model. As introduced in Section 5.2.3, the goal of the recommendation generator is to connect names and types found by the text extractor. Thereby, instances can be suggested to the user that are missed in the input model. In general, these recommendations are independent of the model. This independence allows the approach the identification of missing elements. However, the information of the model can be used to identify common types of the model. The identification of model types helps the approach to separate between multiple possibilities. For example, a recommendation with a model type is more probable to occur in a model of that style than the same recommendation with a type

that is not contained in the model. Additionally, the recommendation agent can search for names in the model elements. If recommendations with the names are build or found, the recommendation agent rates them higher. The reason for this is that model names that occur in the text indicate consistencies. Therefore, they should be used to build recommendations. Otherwise, they would not be considered for the link building. Note that, the recommendation agent does not create links; the agent just uses the information of the underlying model. Moreover, the approach should be able to suggest possible (goal) model elements to the user that does not have a unambiguous type. Therefore, it is needed that *RecommendedInstance*s can be created without the specification of a type.

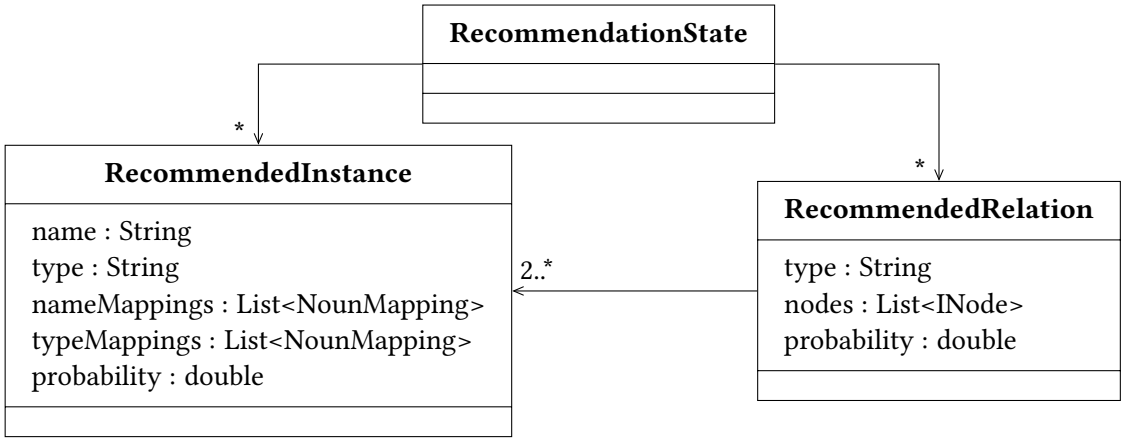


Figure 6.4.: Recommendation Generator

The recommendation generator consists of an agent, holding a state. The *RecommendationState*, as shown in Figure 6.4, stores *RecommendedInstance*s and *RecommendedRelation*s. *RecommendedInstance*s are connections between multiple names and type mappings. The likelihood of that connection is stored in the probability. The name type combination should be picked as representative of the recommended instance. It should be unique. *RecommendedRelations* consist of at least two *RecommendedInstance*s. For the representation of the semantics of that relation the type field has to be set. Moreover, the PARSE nodes that contain the relation (not the end points) are stored.

As usual, the *RecommendationState* holds the logic of the adding and comparison of its elements, while analyzers and solvers invoke the add functionality. Since *RecommendedInstance*s are defined by their unique combination of name and type, the logic behind the adding of recommended instances or relations, is much more easier as at the *NounMappings*. Moreover, it is not important to have a little set of possibly best recommendations. At first, it is important to store all possibilities in the state, instead of having only the best few recommendations. The stored recommendations can be filtered by their probability or fit to user decisions. Furthermore, a big set of possibilities is good for further steps. Especially the connection generator that searches for recommendations that are similar to model elements, profits from a big set of recommendations.

Algorithm 3 shows the *add* functionality for *RecommendedInstance*s in the *RecommendationState*. If a *RecommendedInstance* should be added, the state checks if the corresponding recommended instance is already contained. If the instance is not contained, the state tests whether a previously stored recommended instance has the same name and type. If an existing recommended instance is found, the name and type mappings are extended by the new ones. Since recommended instances are unique by their combination of name and type, the extension is only applied to the identified recommended instance. If neither recommended instance with the same name and type is contained nor a recommended instance with only the same name, the new recommended instance can be added to the state. If the given name appears in the existing recommended instances, it gets more complicated: At first, a loop iterates over all found recommended instances with the same name. If one of them has a similar type, the existing recommended instance is extended by the new mappings. This also occurs, if the existing type is empty, but the new type contains something. As described in the introduction of this chapter, recommended instances do not necessarily need types. Therefore, recommended instances can be stored in the state without types. The reason for their replacement (when adding a same named recommended instance with a type) is that types can be removed afterwards without any disadvantages. If the user rejects a recommendation because of its type, the recommended instance can be suggested with other types. If the user does not accept any name-type combination, it will always be possible to remove the type and the type mappings and recommend the instance without specified type. Therefore, it is a waste of space to store a variation without type. After that, the method checks if there are no mappings that corresponds to the one that should be added. However, the method takes the known name into account. Therefore, no recommended instance with the same name but without type should be added. This is secured by the last if query. If this is passed, the desired recommended instance is stored in the state.

When a *RecommendedRelation* should be added, the state only checks if the combination of participating recommended instances is already contained in any relation. As described above, *RecommendedRelation*s are unique in this combination. If an already stored recommended relation with the same set of participants can be found, it is updated. This adds the new occurrences and recalculates its probability. With a recalculation of a reappearing relation, the probability should increase. If no existing recommended relation can be identified, the new one is added to the state.

6.2.1. Analyzers And Solvers

All suggestions are created by analyzers or solvers of the *RecommendationAgent*. Since analyzers run through the graph (as explained in Section 5.1), solvers only use the existing, available resources. In the context of the recommendation generator level, the analyzers and solvers have access to the PARSE graph, the textual information stored in the *TextExtractionState*, as well as model information stored in the *ModelExtractionState*. All analyzers/ solvers of this agent are *RecommendationAnalyzers/ Solver*s. Based on this information, the recommendation analyzers and solvers build *RecommendedInstance*s and *RecommendedRelation*s.

Algorithm 3 The add functionality of Recommended Instances

```
function addRecommendedInstance(name, type, probability, nameMappings,
typeMappings)

    recInstance ← new recommended instance of the input parameters

    risWithSameName ← all stored recommended instances with the same name
    risWithSameNameAndType ← all stored recommended instances with
                                        the same name and type

    if state contains recInstance then
        return
    end if

    if risWithSameNameAndType is empty then
        if risWithSameName is empty then
            addToState(recInstance)

        else

            for riWithSameName : risWithSameName do
                if riWithSameName.type is similar to recInstance.type ||
                   riWithSameName.type == "" && recInstance.type != "" then

                    extend the mappings of riWithSameName by the new input mappings
                    return
                end if
            end for
            if recInstance.type != ""  then
                addToState(recInstance)
            end if
        end if
    else
        extend the mappings of the one and only element of risWithSameNameAndType
          by the new input mappings
    end if
end function
```

Currently, the *RecommendationAgent* has the following analyzers and solvers:

1. Extraction Dependent Occurrence Analyzer
2. Name Type Analyzer
3. Extracted Terms Analyzer
4. Reference Solver
5. Separated Relations Solver

As described, the recommendation generator is the first layer that is able to use both, textual and model information. For that reason, there can possibly be analyzers that use the model information to extend the textual knowledge base. Therefore, the origin of the knowledge is not traceable to only textual resources, but only related to them.

Extraction Dependent Occurrence Analyzer This analyzer uses model information to find referenced model instances. For every node is checked, whether it could be a name or a type of a model instance. If similar model instance names (or types) are found, they are added as noun mappings of their kind to the text extraction state. This analyzer should run before any other analyzers or solvers are processed, since it extends the underlying knowledge base.

Name Type Analyzer The *NameTypeAnalyzer* searches for patterns of name and types. If these patterns occur, it creates recommendations out of the combination. In general, the analyzer should not question the results of the text extractor. For that reason, only combinations between names and types (and no types and types, or names and names) are built up to *RecommendedInstance*s. However, the analyzers are able to use norts in exchange for names. The use of norts does not doubt the classification of the text extraction state, since it could not inambigously classify the nort as name or type. The patterns are:

Name Type | Nort Type
: A type follows a name: An example is: “For this, the ai component is used”. In this case, *component* is stored as type by the text extractor, while *ai* has been classified as a name. The combination of them (*ai component*) would be created and stored in the recommendation state. In case of a combination of a nort and type, this is done analogously.

Type Name | Type Nort
: A name follows a type: For example: “The layer contains the component ai ”. Since *component* is identified as a type, and *ai* as a name, the combination of them will indicate an instance. Therefore, a *RecommendedInstance* with the name *ai* and the type *component* will be created. This case can be transferred analogous to a type nort combination.

Extracted Terms Analyzer This analyzer uses terms of the text extraction state to create recommendation names and types of multiple words, like *car wheel* of type *common component*. Algorithm 4 shows the search function of the analyzer. If the current node is not included by a term mapping, it is irrelevant for this analyzer. If it is part of a term, the term has to be identified. Since noun mappings have references on nodes (and nodes can be part of multiple, different noun mappings), the search for terms (starting from the node) can result in multiple, different terms. Therefore, the search iterates over every term, to create all possibly referred recommended instances. With the knowledge of the current term, the kind is set. Since terms are only set by analyzers that define them as *name* or *type*, they can not be *norts*. Therefore, the corresponding name/ type field of the creating recommended instance is reserved. To find a possible other part for the recommended instance (type or name), the function looks at the adjacent nodes of the term. If any noun mapppings contain the nodes, they are used to create recommended instances. Furthermore, the analyzer checks if any terms contain the mappings and if these terms are adjacent to the current term. It is secured that the whole term (or similar parts) occur in the text in correct order, directly before or after the current term. If this is the case, a recommended instance is created for every term combination. The algorithm secures that sentences limits are not crossed for finding adjacent noun mappings, nor for identifying adjacent term mappings. In the case that no adjacent noun mappings can be found, the algorithm checks if the term is of kind *name*. If this is the case, a new recommended instance can be created with just that term as name.

Example: Let "The common component car wheel is part of ..." be a sentence in our given documentation. Let *car wheel* and *common component* be term mappings. Then the *Extracted Terms Analyzer* would recognize the word *car* as part of the term *car wheel*. It would check if the whole term occurs at that place. The adjacent words at this term are *component* and *is*, but only the first is contained as noun mapping by the text extraction state. The analyzer creates its first recommended instance. Since the term *car wheel* is a name, and *component* is a type it creates the recommended instance *car wheel* of type *component*. The name mappings are taken from the term. As type mapping, only the *component* mapping is entered. Then, the analyzer checks if the adjacent nodes (in this case *component*) are part of a term mapping. Thereby, the *common component* is found. The search function checks whether the term occurs complete and in correct order before *car wheel*. Since it does and the term *common component* is of kind *type*, a recommended instance of both terms is created: *car wheel* as name and *common component* as type. The mappings are taken from the corresponding terms. Thereby, the process generated two recommended instances. If the user would reject the more specific one, the approach is still able to suggest the instance *car wheel* with a more general type.

Reference Solver The *ReferenceSolver* combines information from textual and model resources. As textual resources the noun mappings of the text extraction state are used. Their reference represents a unified term for all words stored in the mapping. As model equivalent, instances from the model extraction state are used. An instance includes a longest name that should be most descriptive and unique in the model. For better identification, parts of this longest name are stored in a list.

Algorithm 4 The functionality of the extracted terms analyzer

```
function searchRecommendedInstancesForTerm(node)
    termsWithNode ← all stored terms of the text extraction state with that node

    if termsWithNode is empty then
        return
    end if

    for term : termsWithNode do
        adjNounMappings ← Noun Mappings before and after the term
        if adjNounMappings is empty && term has kind NAME then
            create a new recommended instance with just a name
        else
            create a new recommended instance for every adjNounMapping
            adjTerms ← Term Mappings before and after the current term
            create a new recommended instance for every adjTerm
        end if
    end for
end function
```

The solver searches for mentions of model instances in the text. For this purpose, it iterates through the instances of the model extraction state and searches in the text extraction state for name noun mappings with similar references. For this reason, the list of names of the instance is compared to the reference of the noun mapping. When the number of names exceed the predefined threshold, a noun maping is seen as a match to the instance and a *RecommendedInstance* is created. The idea behind this is to generate all possibilities. This enables their usage for further steps or even as recommendation to the user. Differences concerning the correctness of these can be done via the variation of the *RecommendedInstance*s probability. If no similar name noun mapping is found, the solver searches for a name noun mapping, whose reference is similar to the longest name of the instance. If a name noun mapping is similar to the longest name, it is used for the creation of a *RecommendedInstance* (without a specified type). Otherwise, the search continues. In this case, it is searched for names of the instance (parts of the longest name respectively). If this search is successful, the probability is spread over the matches. Thus, all have the same probability. The more matchings are found, the lower is the probability of each resulting *RecommendedInstance*.

Separated Relations Solver The *SeparatedRelationsSolver* is a solver that creates *RecommendedRelation*s. Sometimes, punctuation marks occur in architecture documentations to specify the location of an element. For example: “The component c is contained by layer.ai”. In this case, the *component c* is part of *ai* that lies in *layer*. In context of this work, these specific punctuation marks are called separators. They can be defined in a configuration file. Currently, it contains the punctuation marks ., ::, or :. To represent this implicit relation between the separated elements, this solver runs through the already

found *RecommendedInstance*s. For each, the solver checks if an included name noun mapping contains an occurrence with a separator. If some occurrences are found that contain a separator, they are split into different partial strings. Then the solvers tries to identify the part that is similar to the name of the examined recommended instance. Since the noun mappings are added to the recommended instance by similar naming, and occurrences are added to noun mappings by similar mentions, at least one part of a separated occurrence has to be similar to the name of the recommended instance. For the other parts, corresponding noun mappings are searched. This search can result in multiple possibilities. To provide every possible combination the cartesian product from the found matchings is built. The solver maintains the order of the parts.

Example: The recommended instance *ai* has to be checked. Its list of name mappings contains just a single noun mapping, also called *ai*. In this noun mapping an occurrence *layer.ai* is contained. The solver divides the occurrence into two parts: *layer* and *ai*. The solver checks which of both is similar to the name of the recommended instance. This is *ai*. Then, the solver searches in the text extraction state for noun mappings with the reference *layer*. The solver finds three of them: *lair*, *laser*, and *layers*. For every possibility, the solver creates a recommended relation. Therefore, the relations (lair, ai), (laser, ai), and (layers, ai) are results of that occurrence.

As the example shows, misspellings and different namings can be balanced with a wider definition of similarity. Of course, more possibilities will occur the more loose the similarity is treated. In future work, the similarity could influence the probability of the recommended relations, such that more correct relations are proposed to the user, but every possibility is still available.

6.3. Connection Generator

The goal of the connection generator is to connect (where possible) recommended instances and relations to model instances and relations. As Figure 6.5 shows, these connections are stored in *InstanceLink*s and *RelationLink*s in the connection state. Every link connects exactly two instances or relations. Thus, every link represents a trace link, but in contrast to them, here, the links have additional probabilities.

Currently, links are used as trace links with an additional probability. The probability contains the likelihood of the connection. To be able to offer multiple suggestions to the user, multiple connections of one model element can be stored in the state.

the reason behind the choice that multiple trace link related connections of one model element can exist to the same time lies in the meaning of trace link. A trace link is an unambiguous connection between a textual and a model element. Therefore, there canno be multiple trace links per model element according to the proper meaning of the word *trace link*. I see the term trace link as just a view point that only sees one possibility of many: The one that is most likely so that someone would assure that connection. If or when this assurance takes effect is dependent from the similarity. Since this similarity is represented by the probability, the decision whether a connection is a trace link or not depends on the probability. This decision can be broken down to a threshold.

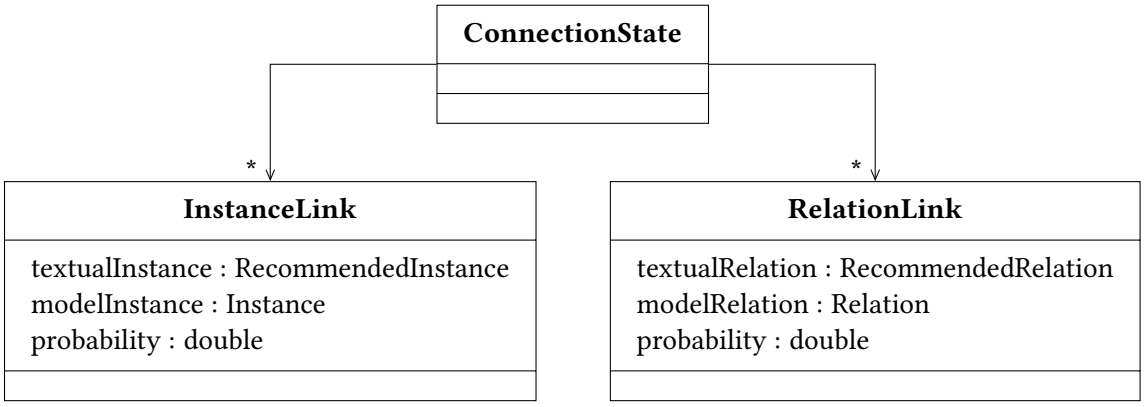


Figure 6.5.: Connection State

Therefore, links of the connection state can be seen as the whole view. For suggestions the view can be reduced to a view point: A single connection per model element. Since links are unique by their combination of textual and model elements, the comparison and addition of links is very simple.

If a new instance link should be added, it is checked if the combination of recommended instance and (model) instance is already contained by any connection of the state. If this is the case, the already stored recommended instance is extended. This work is done by adding the new name and type mappings to the existing mappings of the recommended instance. If the model instance is not already used, a new instance link is created.

The addition of relation links works analogously.

6.3.1. Analyzers And Solvers

The analyzers and solvers of the connection generator have access to all previous states and information. Since analyzers run through the textual graph, they are less common than solvers that look up the states. Even if currently no analyzers are implemented on this level, the abstract analyzer is provided. Thus, future work can simply add on.

1. Instance Connection Solver

2. Relation Connection Solver

Instance Connection Solver This solver connects recommended instances and model instances. For this, it runs through the model instances of the model extraction state. Then, it runs over the recommended instances of the recommendation state to find suggested instances with a similar name. For the recommended instances with the most similar name, an instance link is added to the connection state. If there are multiple, most similar recommended instances, links are created for all of them. For recommended instances without a specified type the probability is decreased. Thus, instance links that include a type specification are preferred.

Relation Connection Solver The relation connection solver works similar to the instance connection solver. Instead of connecting instances it connects relations. Therefore, it searches for every model relation a corresponding recommended relation. If both relations have different amounts of participating instances, the search is continued with another recommended instance. If the amount is the same, the instance references have to be compared. For the recommended relation under examination all instances of its participants are collected that are most similar to the instances of the model relation. Thereby, multiple combinations of relations with the most similar participants are created. If for every (model) instance participant exactly one corresponding recommended instance exists, the relation link between both is added to the connection state. The order of the participants is matched to the model relation. If no match can be found, it is tried to find a recommended relation that has the participants in the wrong way round.

6.4. Configuration

This thesis provides a first step in the direction of suggesting missing elements and trace links to the user. To support future experiments and gain insights on how the collaboration of different analyzers and solvers works, the run configurations can be set in a configuration file. The tables (Table A.1 to Table A.5 in the appendix) provide an overview about the configurations that can be set per class. The defined configurations are loaded automatically as parameters to a final class (*ModelConnectorConfiguration*). From there, every other class can access and load the configured settings. The type columns of the tables specify the type that is available internally.

7. Evaluation

In this chapter, I will present the evaluation of the approach. The structure is oriented to the goals of this thesis and their questions. The goals are:

1. Represent the concept of different graduations of links between model elements and textually described elements,
2. Recognition of textually described model elements without knowledge of the model,
3. Recognition of model elements explicitly named in the textual resource,
4. Identification of links.

The first goal refers to the structure and representation of the links in the approach and is discussed in the first section of this chapter (Section 7.1). The next goals require a data base to evaluate against. Therefore, Section 7.2 describes the creation of a gold standard. Section 7.3 introduces multiple metrics to measure the data. In Section 7.4 the gold standard and metrics are used to evaluate the recognition of model elements without knowledge. The recognition of explicitly mentioned model elements is discussed in Section 7.5. In the last section (Section 7.6), the connections between model and textual elements are examined.

7.1. Representing Links

The first goal of this thesis is to propose a structure that is able to represent a linkage between model elements and textually described model elements. The main question is: Is the chosen representation of links suitable for the problem. To answer this question and discuss the implementation of links, I will summarize what links are and how they are represented in this thesis. A more detailed description can be found in Chapter 4, Chapter 5, and Chapter 6. An overview of the types and their relations is given in Figure 5.8.

As Figure 7.1 shows, the approach is based on elements extracted from the documentation and model. The approach tries to connect them with links, depending on their similarity. In Chapter 4 I divided links in two subgroups: Hint links and trace links.

Trace links are links that connect exactly one textual element with one model element. The connection represents the equality of both elements. The connection is very probable, thus it can be seen as a secure link. In Figure 7.1 they are represented as the lower connection between text elements and model elements. In contrast to trace links, hint links do not have to be as probable. They are subdivided in two more subgroups. The first subgroup are hint links that connect text elements and model elements. They only

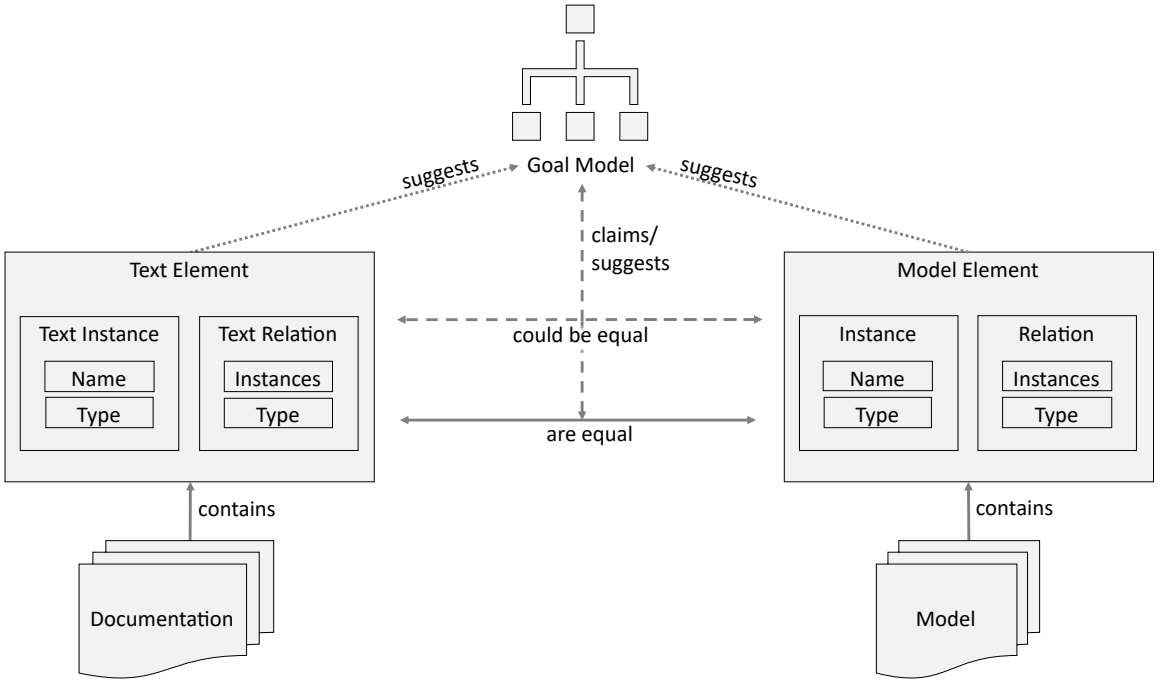


Figure 7.1.: Approach Overview

indicate similarity between text and model elements. They can be transformed to *trace links*, depending on the trace link requirements (e.g. a threshold for the probability). This kind of hint links are represented as *instance links* and *relation links* in the implementation. The second subgroup of hint links are text elements that can not be connected to any model element, but are probable to be in the goal model. Thereby, they suggest additional (to the base set of the model elements) goal model elements. In the implementation they are represented by *recommended instances* and *recommended relations*. Both are preliminary steps of links, but if no corresponding model element can be identified, they can be seen as recommendations that should occur in the model. However, some recommendations with too low probability or number of occurrences should be sorted out. With this underlying the construction, it is possible to define hint links and trace links.

The question *"Is the chosen representation of links suitable for the given problem?"* is subdivided in multiple more specific questions:

1. Does the representation contain at least one textually described element and at least one model element?
2. Can the connected elements be tracked back to their respective occurrences?
3. Does the representation contain unnecessary information?
4. Does the representation enable different graduations of links?
5. Is there a way to suggest textual elements that have no corresponding model elements?

Does the representation contain at least one textually described element and at least one model element?

Since the approach divides links in multiple subgroups to represent them, the specific questions about the structure of links target different representations. The first question is directed to the concept of *instance links* and *relation links*. It scrutinizes the ability to represent trace links and hint links. Both kind of links are constructed to connect one recommended element and a model element. A recommended element is a collection of textual names and types (as text instance or text relation shown in Figure 7.1). Therefore, by construction, links always contain one textually described element and one model element. Since hint links represent possibilities, the approach can represent multiple options for one model element. Instead of creating a new kind of links, the approach represents these opportunities by storing multiple possible links to the same element.

Can the connected elements be tracked back to their respective occurrences?

This aspect is important, since the approach should support the user. By backtracking, the user is able to understand recommendations or verify/ reject parts of them. Moreover, the indications what should be in the goal model or what is missing could be tagged in the text. The approach enables this by construction. As described above, links consist of model elements and recommended elements. Model elements have a unique identifier (*uid*) and can thereby be back tracked to the specified element in the model. The recommendation element, consists of *names* and *types* that are implemented as *noun mappings*. *Noun mappings* unify multiple words of the same meaning. The *occurrences* of every word, as well as the PARSE *nodes* are stored in it. With the PARSE graph, the position in the input text can therefore be restored. With this construction, the question can be affirmed.

Does the representation contain unnecessary information?

For me, unnecessary means that no information is stored that is neither needed for creating links, nor helpful for the decision of the user to reject or verify the connection, nor usable to back track the links to the source. With this definition of necessity, I will discuss the representation presented in Figure 5.8. To fulfill the requirement, every part of the link has to be necessary. Since links are based on textual and model resources, I will start there.

The *model extraction state* contains *instances* and *relations*. *Instances* represent the objects of the model that are connected with relations. They own a *longName* and a *longType* that should be unique. The approach uses the long versions for the textual identification of an *instance*. Beyond that the *longName* can be useful for the user to see which *instance* was meant with a suggestion. The *uid* enables the backtracking to the specific model instance. The list of *names* and *types* gives the possibility to define alternative names. Currently, the *longName* is split (since it usually consists of multiple parts) and stored in these fields. In contrast to the *longName* or *longType*, the shorter *names* and *types* are more practical when searching for references in the textual source. Therefore, I summarize that every stored information in *instances* has its use. Moreover, the *model extraction state* is able to store *relations* that can be used for *relations links*. A *relation* consists of at least two *instances*. These are linked directly. Additionally, they have a *type*. The type is represented as a string, since different models can be used with the approach. The *type* declares the

kind of connection and can further be used as semantic name. Like instances, *relations* own an *uid* that enables the back tracking to the source. Therefore, also *relations* do not store unnecessary information.

Links consist of a model and textual element. The textual element is represented by recommended elements consisting of mappings. These mappings are directly extracted from the text. The extracted information can be represented in three different structures: *Noun mappings*, *term mappings*, and *relation mappings*. Each *noun mapping* is a collection of PARSE nodes with the same meaning. The collected nodes are stored to refer to the source. The approach uses the nodes to examine the context of the mapping. The different occurrences of the unified meaning that is stored as *reference* can be found in *occurrences*. The *occurrences* can be helpful to get an overview what the mapping contains and what has been classified to be the same. Moreover, the approach uses the *occurrences* for some matching algorithms, since the *reference* is not always representative. Additionally, *noun mappings* have a *mapping kind*. This kind specifies whether the mapping is a *name*, a *type*, or a *nort* (name_or_type). The *kind* preselects the data for further steps and helps them to use the *noun mappings* uniformly and correctly (in context of their textual environment). The *probability* encapsulates the certainty that the mapping has that kind. The probability enables the challenging of an already classified mapping and offers the possibility to evaluate comprehensively a probability for the eventually resulting link. In contrast to *noun mappings*, *term mappings* are not directly used in *recommended instances*, but for the identification and connection of terms that consist of multiple words. If a *term mapping* should be used in a *recommendation instance*, its *noun mappings* are extracted and used instead of the *term mapping* itself. Thereby, only *relation mappings* remain on this level as representation of textually described relations. A *relation mapping* consists of at least two *noun mappings*. Additionally, a *preposition* can be set as a node. The *preposition* offers the possibility to refer to a semantic or direct the relation. If the field is not needed, it can be left empty. Moreover, every *relation mapping* has a *probability* that represents the certainty that the contained *noun mappings* are participants of this relation. As already mentioned, this can be used for the calculation of the certainty of links based on this *relation mapping* or just as graduation.

The next level, built on textual elements, is the *recommendation state*. The state has two structures for the representation of recommendations: *Recommended instances* and *recommended relations*. Both contain elements that could be in the goal model. Therefore, the *recommended instances* usually combine *noun mappings* as names and types, but *recommended instances* can also consist of only a name. The names and types are not only stored as *noun mappings*, but also as strings for better comparison. Therefore, the *name* and *type* fields contain a (hopefully) representative term. The approach uses them to connect similar *recommended instances* and *instances* (respectively *recommended relations* and *relations*). The *probability* of a *recommended instance* encapsulates the probability of the textual instance to occur in the goal model. This information is necessary to select recommendations that can not be connected to a model instance. Like every relation representing structure in this approach, the *recommended relation* refers to at least two instances of its level. The *name* of a *recommeded relation* is a semantic name for the underlying relation and can help the user to understand what relation is meant. The textual source of the *recommended relation* is stored as at least one *relation mapping*. This

enables to back track the source and the certainty of the recommended relation. Like for recommended instances, the *probability* of *recommended relations* values the likelihood of that relation to occur in the goal model. Additionally, the *recommended relation* has a list of PARSE nodes. These *nodes* refer to the occurrences of the relation. They contain the nodes, mentioned in the underlying *relation mapping*. Via copying them into another list on the recommendation level, they can be extended without falsifying the textual information. Furthermore, in contrast to the stored node in the *relation mapping* (that are extracted without knowledge of the model), they represent the set of found occurrences with the knowledge of the model.

Since *instance links* and *relation links* only consist of connections between a *recommended instance* and an *instance* or a *recommended relation* and a *relation*, as well as a *probability* that contains the certainty of this connection, I conclude the question: No unnecessary information is stored in links.

Does the representation enable different graduations of links?
As described before (e.g. in Chapter 5), the links can be represented as *recommended instances/ relations* or *instance/ relation links*. *Instance* or *relation links* have a probability that enables a graduation between multiple possibilities. Since recommended elements are not connected to model elements, they can also be seen as a graduation of links. Therefore, the question can be answered with yes.

Is there a way to suggest textual elements that have no corresponding model elements?
Yes. To suggest textual elements to the user that have no corresponding model element, *recommended instances* or *recommended relations* can be used. Their probability is their certainty to appear in the goal model.

As a result, I conclude that this approach offers a structure that is suitable for the problem of identifying missing and common elements between a given textual document and a model.

7.2. Creation Of A Gold Standard For Evaluation

Since the target of the approach is to support a user with identifying missing or common elements between a textual architecture documentation and an architecture model, the gold standard was created in a user study. Thereby, the aid in sense of common answers can be measured in comparison to humans.

7.2.1. Preparation

The study is based on the architecture documentation of the open source project TEAMMATES [29]. TEAMMATES is a cloud based service for managing peer reviews. As usual for open source projects, the documentation sometimes describes not the architecture itself, but packages and processes that are used in the software. In comparison to other open source projects, the documentation of TEAMMATES is comprehensive and in a good shape. As model, I used a figure from the documentation. This model shown in Figure 7.2 contains the main components with their sub packages. The plotted relations represent dependencies between the elements. Since the model extractor is currently not implemented, I manually transmitted the figure in the model extraction state as it could be represented in a PCM. Table 7.1 and Table 7.2 give an overview how model elements are represented in the model extraction state. Every main component is represented with a composite component. Every composite component has at least one interface. The packages are represented by basic components. For the naming of packages, I decided to use only the second part of the name (e.g. *exception* instead of *common::exception*) as the longest name. The lists of partial names and types are generated automatically when creating new instances. For the user study, I numerated the instances and relations. The relations include three types. The relation *in* specifies that a package is in a component. The relation *provide* represents the relation of a component/ package and its interface. Finally, the relation *uses* represents the dependency of an element to another. Thereby, the relation is not connected to the target itself but to its interface.

Long Name	**Long Type**	**uid**
Common	composite component	0
Common Interface	interface	1
util	basic component	3
exception	basic component	4
...	...	...
Storage entity Interface	interface	19
...	...	...

Table 7.1.: The instances of the TEAMMATES diagram in the model extraction state.

Instance 1	**Instance 2**	**type**	**uid**
0	1	provide	10
2	0	in	11
4	19	use	12
...	...	...	...

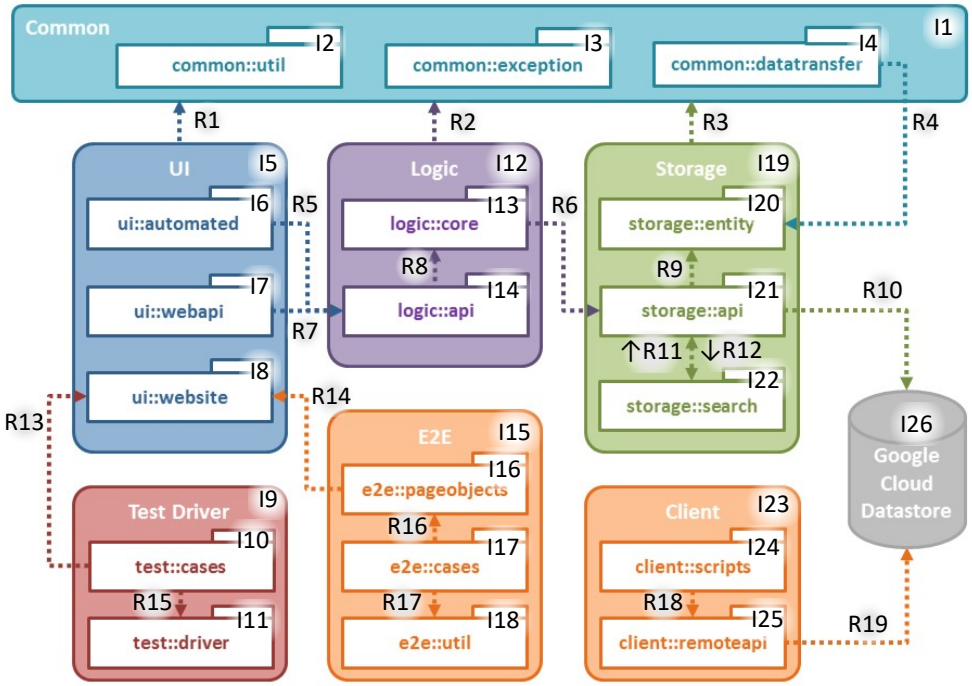


Figure 7.2.: Diagram of Teammates used for the evaluation [29].

Table 7.2.: The relations of the TEAMMATES diagram in the model extraction state.

Since the documentation contained multiple figures, enumerations, subtitles, and marked terms that could disturb the preprocessing, I cleaned the documentation. For this reason, I formulated rules and applied them to the documentation. The resulting text was used to create the gold standard, as well as to run the approach on it. For the user study, I numbered the sentences. The rules that were applied are:

1. Remove headlines
   a) If this is not possible, since the enumeration gets completely unclear: Include the headline in the next sentence.
   b) If possible remove also headlines that are not done with markdown.
2. Remove enumerations
   a) Replace enumerations that consist of a single term with a phrase separating the enumeration items with commas.
   b) Continue introducing sentences for that enumerations if they end with a colon.
   c) Remove “Notes:” headlines
3. Remove tables.
4. Remove links.

5. Remove imports e.g. images.
6. Remove styles like bold type or marks.
7. Remove not functional punctuation marks.
8. If possible remove brackets.
9. Replace explanations that include a colon with a sentence, if necessary.
    a) If a reference or keyword occurs in an explanation replace the first occurrence with the keyword before the colon.
    b) If the colon occurs in a sentence, the sentence can be changed minimally.
    c) Otherwise, add another sentence to the documentation.
10. Replace previous enumerations with a sentence if necessary.
    a) Or extend the text with a chronological sequence, if this is more intuitive.
11. Check the meaningfulness of the text and extend them or add sentences if necessary.
    a) Change headlines to "To [headline] the following information is presented."
12. Replace references to images with a explicit mention.
    For example: "As in the image ..." instead of "as follows".

7.2.2. Concept Of The User Study

The user study includes two parts. The first part contains tasks that are done without knowing the model. The tasks of the second part are done with it. At the beginning, I collected general information about the participants, such as age, study course, grade, and experience. For the answers the questionnaire offered free text fields and yes, no, or no statement boxes for checking.

Part 1

After filling the general questionnaire, the participants got a couple of documents for the first part: The first file introduce the tasks and explains them with examples. The part is partitioned in two tasks: One for the elicitation of instances and the other for relations.

The first part of the first task is to mark all name and types in the text about TEAMMATES. The participants should mark items that they would include in an architecture model. To express their certainty, they were able to mark in three different colors:

1. blue-green: neither agreement nor disagreement that this name/ type appears in the model.
2. dark-green: agreement that this name/ type appears in the model.
3. light-green: fully agreement that this name/ type appears in the model.

The scale is based on the Likert scale. I omitted the negative part of the Likert scale.

After marking all names and types in the text, the participants were instructed to insert all found names in a table (as shown in Table 7.3). Names that belong to the same instance should be collected in one row. Otherwise, they should write them in different rows. The task description explicitly allows multiple mentions of a name in multiple rows if the same name belongs to different instances.

The third part of task one is to extend the table entries by associated types. If no type could be found, the participants should leave the cell empty. If multiple associated types could be found and they semantically mean the same, the participants should list them in one cell of the table. The participants should copy the row if multiple associated (semantically different) types could be found. Thus, one row for each semantically different type would exist.

The last step of the first task is to check the agreement columns for each name-type combination. The confidence should be indicated the agreement for the name-type combination. If a type field would be empty no confidence should be entered. For a clearer delimitation to the marking step, I added a note that the confidence is independent of the decision whether the instance should be in the model or not.

Names	**Types**	**Neither agreement nor disagreement**	**agreement**	**fully agreement**
...	...	...	...	...

Table 7.3.: The instance table of the first task of the user study.

Moreover, a comprehensive example was given to the subjects. It includes all partial tasks and explanations about decisions imaginary participants have made. Moreover, the example shows that multiple possibilities exist to fulfill the task.

The second task of the first part is to fill a table (as shown in Table 7.4) with relations between the found instances. For every relation a semantic name and the participating instances should be entered. If a relation occurs with different instances, it should be inserted additionally in a new row for every constellation. Moreover, sentence numbers should be added, where the relation occurs in the text. The confidence specification is similar to the previous task. For this task only a short example is offered. Since the task is similar to the first task the example consists of an example phrase and a possible solution table.

	Semantic name	**Occurrences (in sentence numbers)**	**Neither agreement nor disagreement**	**agreement**	**fully agreement**
Rel. 1	...	...	...	...	...

Table 7.4.: The relation table of the second task of the user study.

Part 2
Like the first part the second part includes two tasks. The first task is to mark all names and types in the text about TEAMMATES that are contained by the model. The further proceeding is referenced to the procedure of the first task. Like in the first task, a table of instances should be filled. In contrast to the first table, additional columns should be filled: A column with the number of the instance (given by the model as in Figure 7.2) and one with the sentence numbers where this instance occurs. The second task of this part is to write all relations that are contained by the model in another table (like in Table 7.4). The only additional column to fill is the number of the relation that is provided by the model file.

7.2.3. Execution Of The User Study

The user study was executed from home. Since the Corona virus cause a closing of all student working places at the university, a local evaluation was not possible. Therefore, I invited every participant to a Discord channel, to be available for questions and lower the bar for inhibition for them. Since all participants were German, the study instructions were in German. Three participants participated in the user study. All of them were studying computer science: Two in a master's degree programme and one bachelor candidate. The master candidates were in the fourth and sixth semester, while the bachelor candidate was in its eighth. All of them stated that they had heard and passed the software engineering module (Softwaretechnik 1) at the KIT. Moreover, two of them specified that they have experience with models in software engineering, while the other checked the *no statement* box. After the participants filled the general survey, they got the files of the first part. It turned out that some participants overlooked the example on the second page of the instructions or the separate solution work files. Luckily these confusions could be clarified at the beginning of the study during the first task. Beyond that, no questions were asked about the tasks. After the participants ended the first part, they got the second part of the evaluation documents. The time for the first part varied from 1:45h up to 2:35h. Since the user study was planned from 10:00am to 12:00 one participant took a pause during the second part. In general, the time for the second part varied from 0:40h up to 2:35h. Some oral feedback indicated that the participants could have been faster, if the study had been on paper.

7.2.4. Creation Of The Gold Standard

The last step of the creation of a gold standard was to aggregate the answers of the user study. Therefore, I collected the statements of the participants. For the marks in the documentation files, I rated the certainty with points: One point for neither agreement nor disagreement, two points for agreement, and three points for fully agreement. Hereby, it stands out that one participant had a tendency to fully agree, while another never used it. For every word in every sentence I summed up the points of the participants. Furthermore, I accumulated the amount of participants that marked the word. Therefore, I got an overview for every word consisting of the points of each participant, the sum of points for the word, and the amount of participants that marked it. The text has a scope of about 200 sentences.

For the instance table of task one, I aggregated the multiple tables of the participants. Then I summarized rows, if they have a similar name and the same type. Since some participants got the task wrong and inserted German terms, I decided to treat them similar to the English translation. For every row, I summed up the points and the amount of participants that mentioned that name-type combination. Since rows with only a name does not represent a fully instance, I rated those instances with one point. This represents the procedure of the approach. The suggestion without a type should always be the last opportunity presented to the user. Moreover, the task seems less clear than anticipated. Some participants understood *associated types* as types they would use when modeling the instances, instead of types that are given by the text. In this table multiple agreements between the different participants exists.

In contrast to the instance table, the aggregated relation table for the second task was mostly a composition of the single tables. Only one agreement between participants exists. The other relations all have different terms of the participating instances. Therefore, I could not summarize multiple rows, since the participating instances are unique feature of the relations. Moreover, this shows that finding relations without the knowledge of an underlying model is highly disambigious, even if the participants have general knowledge of models. Thus, this table was not used in the further evaluation.

Since the instance table of the third task contains references to the specific instances of the model, the aggregation of the different entries were much easier. Only if the type differed, multiple rows were created. In case of one participant the names of the instances were not inserted in the table. I compensated this fault by entering the name of the instance given by the model. Since the gold standard should only contain one solution per instance, for each instance the row with the highest points were taken. As a result, all instances, except of one had more than four points and at least two participants that specified it and assigned it to the same instance.

The table of the relations of the fourth task could be summarized as well as the instances of the third task. In contrast to the instance table, I aggregated all relations of one specified model relation. This was possible, because the names of the instances always were similar. Therefore, the aggregated table did not contain multiple variations for a model relation and was unambiguous. A failure in the user study was to not explicitly name the *in* relations in the model, since these are the only relations that are currently identified by the approach. Thus, the gold standard is only an indicator what could be found additionally. The table contains 19 relations. This is about 10% of the sentences in the text. Therefore, more analyzers for relations should be added in future work.

Since the probabilities in the approach are not balanced yet, all results are unambiguous and not rated. Moreover, for this first approach it is more important whether the results of the approach contain the correct solution instead of whether it has a high probability. Therefore, the approach (the classifier) just decides between a true or negative value. Since only selections (positive results) are presented by the approach, negative values include all not selected values.

In contrast to the results of the approach, the gold standard offers the possibilities to select values depending on the agreement of the participants. In the user study the agreement was expressed by a scale from one to three points. In the evaluation, multiple sets will be considered. The first section contains all marked elements. That means that

this set contains all elements that someone has been seen as a possibility to be correct. The second section contains all values of the reference that have more than two points. This is the case, if someone fully agreed or at least one participant agreed with the value and another was ok with it. The last set contains all values that have more than four points. This represents values, where at least one person of two persons fully agreed with it and the other agreed, or two of three agreed and the third one was ok with it. In any case the two participants agreeing would outvote the third one. Therefore, the goal is to include all of these values.

7.3. Metrics

In this section, the metrics that are used in the further evaluation are described. The following paragraphs describe the metrics precision, recall, F1-Score, miss rate, and fall-out. For their calculation the classification results are compared to reference values.

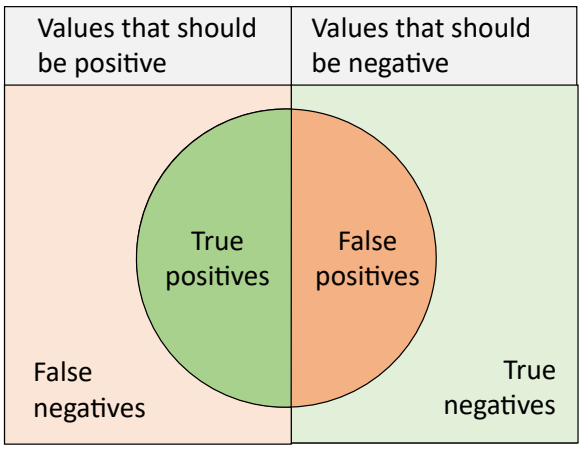


Figure 7.3.: Model of a confusion matrix

Figure 7.3 shows the model of a confusion matrix relating the results of a classification. The columns represent the classification in the reference. In this case these values are the results of the gold standard. The values of the circle are the results that the approach classifies as positive. The surrounding of the circle are the negative results of the classifier. In case of this approach, these are all values that are not selected. In the evaluation of the text extraction this would include all words that are not recognized as names, types or norts. True positives are correctly classified values. In contrast, false positives (fp) are results of the classifier that are classified as true, but not marked as true in the reference. Thus, these are faults in the classification. Another fault group is called fn (false negatives). The false negatives include values that should have been classified as true (since they are true in the reference), but are marked as false. The remaining group that is needed for the rating of classifications are true negatives (tn). These are values that are negative in both, the reference and the classification result.

Precision Precision describes the share of correct suggestions of all suggestions the approach presented. As I will describe in Chapter 8 there will be many possibilities to improve this metric in the future work. It is defined as correctly positive classified results in relative to all positive classified results.

$$Precision = \frac{tp}{tp + fp} \tag{7.1}$$

Recall The recall of a classification represents the proportion of the correct values in the classification result. Especially in approaches that sort out possibilities (like this one), the recall is more important than the precision. If some relevant information can not be found in the first steps, the information is not accessible for the next steps. The recall is defined as all correctly positive classified results in proportion to all positive results of the reference.

$$Recall = \frac{tp}{tp + fn} \tag{7.2}$$

F1-Score The F1-Score is the harmonic mean of precision and recall. It symbolizes the proportion of correct classified results versus the proportion of found positive values. Thus, it measures a trade off between recall and precision.

$$F1\text{-}Score = 2 * \frac{precision * recall}{precision + recall} \tag{7.3}$$

Miss Rate The miss rate, also known as false negative rate, measures the proportion of missed positive values. In context of this thesis, this value is interesting since it represents the loss of information after a classification. Here, it indicates how much more information should be found and classified correctly and symbolizes the potential of future work.

$$Miss\ Rate = \frac{fn}{fn + tp} = 1 - recall \tag{7.4}$$

7.4. Textual Element Recognition

In this section, the results for the goal *Recognition of textual described model elements without knowledge of the model* are discussed. Therefore, the goal is split in multiple questions:

1. Does the approach recognize textual described elements without any model information?
    a) Does the approach recognize the names and types of elements that are included in the text, without knowledge of the model?
    b) Does the approach suggest instances and relations that are included in the text, without knowledge of the model?

To measure the recognition of the textual elements without knowledge of the model, the implementation ran with an empty model. As corresponding reference the gold standard of the first part was taken.

The first question (question *a)*) asks for the recognition of names and type of elements in the text. The marking of these were the first part of the first task in the user study. Therefore, the results are compared to these. As described in Section 7.2.4, the results of the gold standard can be divided in multiple sets, dependent of the certainty of correctness. Since the probability functionality is not balanced yet in the approach, the results of the approach are counted as one.

Reference set	Precision	Recall	F1	Miss rate
>4 points	7.18	68.63	13.00	31.37
>2 points	18.46	77.92	29.85	22.08
all	31.39	81.82	45.37	18.18

Table 7.5.: The results of the evaluation of the marks of the first task in percent per reference sets

Table 7.5 shows the results of the evaluation. If all marks are considered, the recall is at about 81% while the precision is about 31%. For the first step of a first approach that is based on a sort out process, this is ok. Only 18% of the reference marks are missed in the results of the approach. More problematic is the small precision if the reference set is chosen by more secure values. However, the question is whether the problem without knowledge of anything of the model is comparable to the task that the approach should do. In contrast to the approach, the participants of the user study had an idea of how an architecture model is structured or how it could look like. The approach has none of this information. Therefore, for future work the measurement of the quality of the approach with knowledge of a meta-model or other schematic information could be interesting.

The second question of this section asks for the comparison of the resulting recommended instances with the defined instances of the first part of the gold standard. The result represents the problem of the previous question combined with the missing knowledge of the model. For the evaluation of the instance table, I decided to calculate only the results for the set with more than four points. Almost all of the values of the user study that were inserted by at least two participants are in this set. This reduces the number of 112 elements to 44 elements and the reference seems to be more realistic. To match these elements three different stages were evaluated. The first state is true, if the name the current instance of the reference is contained by the occurrences of a recommended instance. The second state is true, if the recommendation state contains an instance with one of the names given by the reference as name. The third state is true if this recommended instance has the same type as the reference.

Matching Strategy	Precision	Recall	F1	Miss rate
occurrences	25.58	50.00	33.85	50.00
name	9.30	18.18	12.30	81.82
name + type	2.33	4.77	3.13	95.46

Table 7.6.: The results of the evaluation of the instances of the first task in percent by matching decisions. Only references with more than four points were considered.

As can be seen in Table 7.6, the results are strongly dependent from the matching decision. As described, I think the lack of knowledge is responsible for the bad results. Moreover, the misunderstanding of the participants at this task could have lead to the bad matchings, since they inserted types that were not contained in the text. However, the approach suggests instances even if only their occurrences contain 50% of the names given by the reference. Therefore, the second question can be affirmed and the goal is reached.

7.5. Model Element Recognition

The goal of this section is to verify and measure the recognition of model elements explicitly named in the text. For this reason the second part of the gold standard is taken for the evaluation. The general question behind this goal is subdivided in multiple parts:

1. Does the approach recognize described model elements in the text?
    a) Does the approach recognize the names and types of model instances?
    b) Does the approach recognize the combination of name and type as one instance?
    c) Does the approach recognize the mention of multiple instances in a relation as one relation?

For the first question the tags of the documentation of task three is compared to the results of the approach. Like in Section 7.4, the certainty points of the participants are summed up. Then, the reference is partitioned in sets with different confidence thresholds. For every set the results of the metrics are calculated.

Reference set	Precision	Recall	F1	Miss rate
>4 points	39.85	85.04	54.27	14.96
>2 points	46.13	81.70	58.96	18.30
all	48.71	81.99	61.11	18.01

Table 7.7.: The results of the evaluation of the tags of the third task in percent per reference sets

The results in Table 7.7 show that the approach identifies at least 80% of the names and types given by the participants. If only names and types with more than four points are considered, the recall increases. At this point, the miss rate is under 15%. With an F1-Score of 54% this is a good baseline for the selecting, combining, and suggesting of the further steps.

For answering the second partial question, the instance table of the third task is compared to the recommended instance found by the approach. Since there were mulitple participants, multiple possibilities per instance exist in the table. I decided to summarize the possibilities such that only one possibility per model instance would be in the gold standard. Thus, the possibilities with the highest points (per model instance) were taken. The result consists of 24 name-type combinations with more than four points. With the exception of one case with only three points. Like in the instance table of part one, the types of the gold standard include German terms. The basic components of the theoretical PCM were mostly identified as packages. For the comparison to the recommended instances, I chose multiple matching strategies. This matching strategies define whether a true positive or a false negative is created. The first strategy is *occurrences*. With this strategy, a hit is created if one of the names of the instance is mentioned in the occurrences of any matching. Thus, this strategy represents the existence of the name. It could be wrongly mapped to another name, instance or type. The correct naming is checked in the next strategy. If a recommended instance with a similar name exist, a hit is created. As similar all names were taken that could be interesting for users that would get such a suggestion. For example: *driver* instead of *Test.driver*, *Logic API* instead of *logic.api*, as well as *util/ utility* instead of *E2e.util*. If explicitly substructures (basic components/ packages) like *logic.api* were named in the gold standard, the name of the main structure (e.g. *logic*) was not accepted as a name. This is an important detail when looking at Table 7.8, because the name mappings of the packages were often collected in the name mappings of the component. Thus, the packages were not identified as a single instance.

Matching Strategy	Precision	Recall	F1	Miss rate
occurrences	23.30	96.00	37.50	4.00
name	14.56	60.00	23.44	40.00
name + type	4.85	20.00	7.81	80.00

Table 7.8.: The results of the evaluation of the instances of the third task in percent by matching decisions.

Table 7.8 shows the results of the evaluation. As expected the matching strategy containing the types underperforms, since only components were recognized. The recall of the instance names in the occurrences of the recommended instances is pretty good, even if the recall decreases to 60% when the names are compared. For a first try I rate this as ok. Since recommended instances are in general not the last step of the approach, the low precision can be seen as sufficient.

The last question deals with the recognition of relations in the recommendation state. As described in Section 7.2.4 the table that was thought to be compared to the results of the approach did not include *contain\in* relations at all. Since these are the only relations

that are identified in this thesis, the comparison to the table is not expressive. Instead of comparing the relations to the textual extraction of the gold standard, I compared them to the 18 *in* relations defined by the model. Thereby, the recall is only a lower bound, since it is not verified that every relation of the model could be found in the text.

Matching Strategy	Precision	Recall	F1	Miss rate
rough	50.00	52.94	51.43	47.06
exact	11.11	11.77	11.43	88.24

Table 7.9.: The results of the evaluation of the recommended relations compared to the existing *in* relations of the model, in percent by matching decisions.

Table 7.9 shows the results of the comparison divided in two matching strategies. The *rough* strategy does not need a similar name. Thereby, the strategy allows many faults and should be seen more as values what could be, if the instances were named correctly. The results of the comparison with the actual names of the instances can be seen in the row with the *exact* matching strategy. Here names had to be as similar as in the instance table. The results show that an F1-Score of 51% could be possible if the names of the recommended instances were selected more appropriately. Therefore, the names of the instances have to improve. Currently, when matching the exact names of the instances of a relation only 11% could be reached. But, as described before, these are lower bounds for this documentation and model and could be higher if the comparable set had been generated with the information from the text.

Nevertheless, the goal of the model element recognition is reached. The approach is able to identify names and types of the model with a recall over 80% while the precision is about 40%. Moreover, the approach identifies name and type combinations. The quality of them depends heavily on the matching strategy. Additionally, I assume that it is dependent from the type names of the underlying model, since packages were not declared as model types. Also, recommended relations were found. The lower bound for the F1-Score is about 11%. This should be improved in future work.

7.6. Creating Tracelinks

The goal of the link identification can be formulated as the question: Does the approach suggest textually described model elements and link them to their corresponding elements in the model? To answer the question, I differentiate between instances and relations.

For the evaluation of that question in relation to instances, the gold standard of the instance table of part two is used. Like in Section 7.5, the user study matchings with the highest points per model instance were chosen. Thereby, every instance is described unambiguous by the standard. I compared the instance links of the connection state with the results of the gold standard. This procedure was pretty similar to the one in Section 7.5. Additionally, two metrics were used: The *name + instanceNo* metric checks if the name is correct and mapped to the correct instance number of the model. The other metric (*name + type + instanceNo*) checks additionally if the type is correct. Since the names of the types differ from the ones defined in the PCM, this is not often the case.

The results of the evaluation are shown in Table 7.10. Precision as well as recall are at 80%, if the *occurrences* strategy is followed. If the strategy is changed to the comparison of the names, the F1-Score drops to 45%. But as the matching strategy of *name + instanceNo* shows, 81.8% of them are mapped to the correct instance. If the type is included to the matching, the F1-Score decreases about 17% points at least. As described in Section 7.5, I think that the names of the types in the underlying model are responsible for this. Without considering them, the values especially for the matching by names and instance number has to improve. Nevertheless, the results are promising.

Matching Strategy	Precision	Recall	F1	Miss rate
occurrences	86.96	80.00	83.33	20.00
name	47.83	44.00	45.83	56.00
name + type	26.09	24.00	25.00	76.00
name + instanceNo	39.13	36.00	37.50	64.00
name + type + instanceNo	21.74	20.00	20.83	80.00

Table 7.10.: The results of the evaluation of the instances of the third task in percent per matching decisions.

For the evaluation related to relations the *in* relations of the model were used. Thereby, as explained in Section 7.5, the results represent a lower bound for the suggestions of relation links. The procedure was similar to the one in Section 7.5. Instead of comparing the relations of the recommendation state, the relation links from the connection state were used.

Matching Strategy	Precision	Recall	F1	Miss rate
rough	45.46	29.41	35.71	70.59
exact	18.18	11.77	14.29	88.24

Table 7.11.: The results of the evaluation of the recommended relations compared to the existing *in* relations of the model, in percent per matching decisions.

With the *rough* matching strategy, the precision is similar to the one of the comparison with the recommended relations. In contrast to them, the recall is about 20% points lower, when using the relation links. In case of the *exact* name matching for the instances, the recall for both cases is equal, but when comparing the relations of the connection state the precision is 8% points higher. As described in the previous case (Section 7.5), the values of the second matching strategy would improve, if the naming would be better. If the values of the *rough* matching strategy could be reached, the results would increase to a good level. This is especially the case, since they are only a lower bound.

Since the links are the last stage of the approach and are thereby directly suggested to the user, the results should have had a good mean between precision and recall. In both cases, the F1-Score that represents that mean is too low. However, since the links are created only by two solvers, there is potential to improve their strategies with additional analyzers or solvers. Nevertheless, the approach is able to find model elements and connect them to textual elements.

7.7. Threats To Validity

The evaluation implies multiple threats to validity. Since the user study only used one documentation and similar tasks and tables for the first and second part the history effect could have been occurred. The history represents a learning effect that is caused by repetitions in a study. To avoid it, the second task (with the model) was only handed out if the participant finished the first part. Moreover, the second task was more restrictive and detailed what should be marked in the text or inserted in the table. Therefore, the effect should have no great influence on the second task. However, the text was already known and the novelty effect exhausted. It could have been happened that the participants worked less precisely in the second part. Additionally, an instrumentality effect could have been occurred. Since the user study was done from home and at a computer, the environment of the participants was not controllable. After the study, a participant said that he was not concentrated during the study. Since one participant took a pause in the second part of the study, a maturation may also have effect. In addition to the official effects that may have been happened in the user study, the creation of the gold standard may also contain threats to validity. If some participant only specified the name as an instance, I counted it as one point in the gold standard. Furthermore, some participants got the first and third task wrong and entered German terms as types instead of terms that were contained in the text. I treated them as similar if the German term was the translation of the English term. This could be a threat to the validity, since the German term is not contained in the text. Nevertheless, the semantic were the same and could have come from the text. However, the dimensions of the effect can not be measured. Moreover, some participants did not insert the names of the model elements in the instance table of the second part. Since the numbers of the model instances were given, I entered the names of the model elements in the column. However, the occurrence of these terms in the text is not verified. Nevertheless, only results, comparing the occurrences of the instances are under influence of this effect, since the approach does not use internally the whole names of the model elements.

8. Conclusion And Future Work

In software engineering, consistency between artifacts is an important topic. This thesis provides structures and processes that are first steps to deal with consistency violations between an architecture documentation and its models. The focus of this analysis are instances and relations of the model. The approach suggests links between textual elements and their counterparts in the model. Moreover, it generates recommendations for elements that are not in the model. These recommendations can support a user who checks the consistency of software artifacts.

This thesis differentiates different levels of agents for a modular goal-oriented approach. Since INDIRECT preprocesses the given documentation and adds linguistic information to it, the approach is based on the resulting graph. To remain the structure of PARSE, the agents on the different stages of this approach inherit from PARSE agents. In contrast to them, their gained knowledge departs from the textual basis. Thereby, the new information is not added to the graph as usual, but stored in decentralized states. Since the later agents are dependent from the agents before them, agents have access to the states of every agent that ran before. The most underlying agent is the text extraction agent. The agent is based on the PARSE graph and uses linguistic information to extract and classify nouns from the input text. The goal is to extract terms that could be used as names or types of elements in the model. For this, the agent has multiple analyzers and solvers. Analyzers run through the PARSE graph and transmit the gained information into the state. Solvers look up that state and calculate conclusions. If an analyzer or solver finds relevant information, it transfers them with a confidence to the state. The state decides what is done with this information. Thereby, the approach enables a voting between multiple analyzers and solvers. Moreover, the state merges multiple information. At the text extraction level information is stored as noun mappings. Noun mappings unify different words under one reference. Whether a new information is added to an existing noun mapping or a new one depends on the behavior of the state and the reference. The reference is given by the analyzers when they transfer their information to the state. As result of the first step of this approach the text extraction state includes multiple grouped nouns (as noun mappings) that are classified as a name, type, or possibly both. The next agent building on the knowledge of the text extraction state is the recommendation agent. The goal of the agent is to identify textually described elements. Since the text extraction agent only provides names and types the recommendation agent combines these to gain recommended instances and relations. For this the agent has access to the text extraction state and all underlying information, as well as the model. Moreover, the agent finds instances and relations that are textually described but missed in the model. For a uniform structure, the different analyses are outsourced in analyzers and solvers. The last stage of the approach is the connection agent. This agent builds links between the recommended elements and model elements. The built links are similar to trace links. Furthermore, each links

has a probability that represents the likelihood for the connection. Thereby, different graduations and suggestions for links can be made. The model is accessed by the model extraction state. The state is designed to contain multiple types of models. To provide a uniform interface for different models only relevant information, like name, type, and a unique identifier are stored in the elements. As the other states the model extraction state can be filled by analyzers of the model extraction agent. The implementation of this feature is part of the future work.

The evaluation is based on the open-source project TEAMMATES. TEAMMATES is a software for managing peer-reviews. A user study was made to get a comparable gold standard to the approach. The user study contains two tasks per part. The first part is done without knowledge of the model. Thus, it is comparable to the text extraction and partial recommendation generation of the approach. The second part was done with a model that is contained in the original documentation. The first task of each part is to mark names and types of textually described model elements. The three participants had the possibility to mark the words in three different colors regarding their confidence. In concern of the evaluation the colors were translated into a scale from one to three points. These points were summed up in the gold standard to get graduations of multiple solutions. Furthermore, the first task includes to combine names and types and build instances that should be contained by an architecture model. For every identified instance the participants set a confidence. The second task is similar to the first. Instead of instances relations should be build. The second part is equal to the first, except that the participants have knowledge of the underlying model. Both results of the relation tasks were not used in the evaluation as they were not applicable for it. Instead of them, the *in*-relations of the model were taken for comparison.

The approach identified 80% of the names and types marked by the participants without model knowledge. If only names and types with more than four points are concerned, the approach reaches a recall of 68%. The names of the instances of the gold standard are found with an F1-Score of 33.85% in the occurrences of the recommended instances. This means that the names were identified as names of an instance, but not as reference for it. For more specific matching strategies, like a comparison of the names, the results differ very much. Thereby, the conclusion is that without model knowledge such comparisons are not promising. For the second part of the evaluation the model knowledge was available for the participants and the approach. The approach identifies at least 80% of the names and types of textually described instances. Moreover, the recall even increases with the confidence of the participants: For results with more than four points the F1-Score was 54%. Therefore, the approach has a good base to build recommendations out of it. Also in the combination part of the first task the approach is better if the model is known. In 96% of the cases the names of the instances defined by the gold standard are found in the occurrences of recommended instances. Furthermore, 60% are also used as the name of the recommended instance. The recommended relations reached a lower bound of 11% as F1-Score when the names of the instance have to be similar. When the names are compared only roughly it is possible to reach an F1-Score of 51%. A similar effect is observed in the evaluation of the relation links. When comparing exactly the names of the participating instance links, the F1-Score is at 14%. Otherwise, it is at 35%. Since, the names of the participating instance links are decisive for the matching of the relation, the

results for the identification of instance links show the same: When matching the instance links of the gold standard with the ones of the approach by occurrences an F1-Score of 83% is reached. When choosing a matching strategy that requires similar names, the F1-Score sinks to 45%. In 36% of the gold standard defined instance links the approach matches the names and model instances. The precision and recall are almost the same across the different strategies.

Since this approach is only a first step towards the main goal, there is some future work that need to be done. At first, the model extraction should be implemented for at least one model type (e.g. PCM). Thereby, the approach could be applied automatically. The probability feature is another feature that is also anchored in the approach, but currently not used. By implementing a meaningful probability function the precision and decisions could be improved on multiple stages. Moreover, a reasonable probability function allows the suggesting of recommendations or links in a rated order to the user. Furthermore, the exact influences on the thresholds and thus the results are interesting for a better understanding of the correlations. Maybe the thresholds could be adapted to different documentations to get the best results. For this purpose an automated learning algorithm could be feasible. In general, more analyzers and solvers would improve and extend the results of the approach. Especially respecting further relations, additional analyzers and solvers should be written. For the disambiguation of noun mappings and further structures, I recommend a context based analysis. This can be done with different strategies. One example is a sliding context window that examines the close surrounding of a word. However, a semantic database could also help with the disambiguation. Furthermore, it could be useful to analyze the distribution of noun mappings in multiple sections of the text. Thus, mappings that appear lonely in multiple sections and more frequently in one section could be indicators to have multiple meanings. In general, the frequency of terms could be helpful to evaluate the importance of it. Moreover, the frequency and structure in the text could be related to the structure in the model. To prevent the recognition of common terms as noun mappings (e.g. *system test*), an ontology of these terms could help to exclude them from the classification. Furthermore, the ontology could not only be used for exclude terms, but also to emphasize others. Eventually, a list of general model element types like *component* or *class* could help recognize instances. In general, the matching strategies need to be improved. As the evaluation shows the matching by name could be tweaked. There are two possibilities for improvement. The first option is to revise the naming of noun mappings and further structures. On the other hand, the matching strategies could be revised or extended. Moreover, another evaluation should be done to verify that the approach is applicable in different projects the future evaluation should use other projects. This would be especially important for a more accurate evaluation of the relations. Finally, an open question is how much the exclusion of figures and other graphics affects the results and understanding of the remaining documentation.Instead of formulate the graphics in natural language these are currently excluded. The reason for this are the costs and unexplored benefits of including the graphics. It would be interesting to evaluate the approach with textually described graphics to see if the results differ.

Bibliography

[1] URL: https://corenlp.run/ (visited on 09/22/2020).

[2] Michele Banko et al. "Open information extraction from the web". In: *Communications of the ACM* 51.12 (Dec. 1, 2008), p. 68. ISSN: 00010782. DOI: 10.1145/1409360.1409378. URL: http://portal.acm.org/citation.cfm?doid=1409360.1409378 (visited on 02/17/2020).

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction". In: *Journal of Systems and Software*. Special Issue: Software Performance - Modeling and Analysis 82.1 (Jan. 1, 2009), pp. 3–22. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.03.066. URL: http://www.sciencedirect.com/science/article/pii/S0164121208001015 (visited on 02/14/2020).

[4] Steffen Becker et al. *Modeling and Simulating Software Architectures - The Palladio Approach*. 2016. ISBN: 978-0-262-03476-0.

[5] Sergey Brin. "Extracting Patterns and Relations from the World Wide Web". In: *The World Wide Web and Databases*. Ed. by Paolo Atzeni, Alberto Mendelzon, and Giansalvatore Mecca. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1590. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 172–183. ISBN: 978-3-540-65890-0 978-3-540-48909-2. DOI: 10.1007/10704656_11. URL: http://link.springer.com/10.1007/10704656_11 (visited on 02/17/2020).

[6] Erik Johannes Burger. "Flexible views for view-based model-driven development". In: ACM Press, 2013, p. 25. ISBN: 978-1-4503-2125-9. DOI: 10.1145/2465498.2465501. URL: http://dl.acm.org/citation.cfm?doid=2465498.2465501 (visited on 05/20/2018).

[7] Erik Burger, Victoria Mittelbach, and Anne Koziolek. "View-based and Model-driven Outage Management for the Smart Grid". In: Proceedings of the 11th Workshop on Models@run.time co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016). Saint Malo, France, p. 8.

[8] Kai-Uwe Carstensen et al. *Computerlinguistik und Sprachtechnologie: Eine Einführung*. Google-Books-ID: OiogBAAAQBAJ. Springer-Verlag, Nov. 4, 2009. 750 pp. ISBN: 978-3-8274-2224-8.

[9] Deva Kumar Deeptimahanti and Muhammad Ali Babar. "An Automated Tool for Generating UML Models from Natural Language Requirements". In: *2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009 IEEE/ACM International Conference on Automated Software Engineering. ISSN: 1938-4300. Nov. 2009, pp. 680–682. DOI: 10.1109/ASE.2009.48.

[10] Oren Etzioni et al. "Unsupervised named-entity extraction from the Web: An experimental study". In: *Artificial Intelligence* 165.1 (June 2005), pp. 91–134. ISSN: 00043702. DOI: 10.1016/j.artint.2005.03.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0004370205000366 (visited on 02/17/2020).

[11] Oren Etzioni et al. "Web-Scale Information Extraction in KnowItAll (Preliminary Results)". In: ACM 1-58113-844-X/04/0005. 2004, p. 11.

[12] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. "Semantically Enhanced Software Traceability Using Deep Learning Techniques". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). ISSN: 1558-1225. May 2017, pp. 3–14. DOI: 10.1109/ICSE.2017.9.

[13] Marti A. Hearst. "Automatic Acquisition of Hyponyms from Large Text Corpora". In: *COLING 1992 Volume 2: The 15th International Conference on Computational Linguistics*. COLING 1992. 1992. URL: https://www.aclweb.org/anthology/C92-2082 (visited on 02/17/2020).

[14] Tobias Hey. "INDIRECT: Intent-Driven Requirements-to-Code Traceability". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). ISSN: 2574-1934. May 2019, pp. 190–191. DOI: 10.1109/ICSE-Companion.2019.00078.

[15] Mohd Ibrahim and Rodina Ahmad. "Class Diagram Extraction from Textual Requirements Using Natural Language Processing (NLP) Techniques". In: *2010 Second International Conference on Computer Research and Development*. 2010 Second International Conference on Computer Research and Development. May 2010, pp. 200–204. DOI: 10.1109/ICCRD.2010.71.

[16] Leonid Kof. "An Application of Natural Language Processing to Domain Modelling – Two Case Studies". In: International Journal on Computer Systems Science Engineering, p. 27.

[17] Leonid Kof. "Natural Language Processing: Mature Enough for Requirements Documents Analysis?" In: *Natural Language Processing and Information Systems*. Ed. by Andrés Montoyo, Rafael Muńoz, and Elisabeth Métais. Red. by David Hutchison et al. Vol. 3513. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 91–102. ISBN: 978-3-540-26031-8 978-3-540-32110-1. DOI: 10.1007/11428817_9. URL: http://link.springer.com/10.1007/11428817_9 (visited on 02/29/2020).

[18] Leonid Kof. "Using Application Domain Ontology to Construct an Initial System Model". In: IASTED International Conference on Software Engineering, p. 6.

[19] Max E. Kramer, Erik Burger, and Michael Langhammer. “View-centric engineering with synchronized heterogeneous models”. In: ACM Press, 2013, pp. 1–6. ISBN: 978-1-4503-2070-2. DOI: 10.1145/2489861.2489864. URL: http://dl.acm.org/citation.cfm?doid=2489861.2489864 (visited on 06/05/2018).

[20] Marie-Catherine de Marneffe and Christopher D Manning. *Stanford typed dependencies manual*. Tech. rep. Stanford University.

[21] Mike Mintz et al. “Distant supervision for relation extraction without labeled data”. In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*. ACL-IJCNLP 2009. Suntec, Singapore: Association for Computational Linguistics, Aug. 2009, pp. 1003–1011. URL: https://www.aclweb.org/anthology/P09-1113 (visited on 02/17/2020).

[22] Sabine Molenaar et al. “Explicit Alignment of Requirements and Architecture in Agile Development”. In: *Requirements Engineering: Foundation for Software Quality*. Ed. by Nazim Madhavji and Liliana Pasquale. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 169–185. ISBN: 978-3-030-44429-7. DOI: 10.1007/978-3-030-44429-7_13.

[23] *Natural Language Toolkit — NLTK 3.5 documentation*. URL: https://www.nltk.org/ (visited on 09/17/2020).

[24] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. Cambridge, MA: MIT Press, Oct. 2016. 408 pp. ISBN: 9780262034760. URL: http://mitpress.mit.edu/books/modeling-and-simulating-software-architectures.

[25] Sandra Schröder and Georg Buchgeher. “Applicability of Controlled Natural Languages for Architecture Analysis and Documentation: An Industrial Case Study”. In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2*. ECSA ’19. event-place: Paris, France. New York, NY, USA: ACM, 2019, pp. 190–196. ISBN: 978-1-4503-7142-1. DOI: 10.1145/3344948.3344981. URL: http://doi.acm.org/10.1145/3344948.3344981 (visited on 10/02/2019).

[26] Sandra Schröder and Matthias Riebisch. “An Ontology-based Approach for Documenting and Validating Architecture Rules”. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ECSA ’18. New York, NY, USA: ACM, 2018, 52:1–52:7. ISBN: 978-1-4503-6483-6. DOI: 10.1145/3241403.3241457. URL: http://doi.acm.org/10.1145/3241403.3241457 (visited on 01/18/2019).

[27] Sandra Schröder and Matthias Riebisch. “Architecture Conformance Checking with Description Logics”. In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*. ECSA ’17. event-place: Canterbury, United Kingdom. New York, NY, USA: ACM, 2017, pp. 166–172. ISBN: 978-1-4503-5217-8. DOI: 10.1145/3129790.3129812. URL: http://doi.acm.org/10.1145/3129790.3129812 (visited on 04/01/2019).

[28] Antony Tang et al. “Traceability in the Co-evolution of Architectural Requirements and Design”. In: *Relating Software Requirements and Architectures*. Ed. by Paris Avgeriou et al. Berlin, Heidelberg: Springer, 2011, pp. 35–60. ISBN: 978-3-642-21001-3. DOI: 10.1007/978-3-642-21001-3_4. URL: https://doi.org/10.1007/978-3-642-21001-3_4 (visited on 03/29/2020).

[29] *TEAMMATES/teammates*. GitHub. URL: https://github.com/TEAMMATES/teammates (visited on 09/26/2020).

[30] *Universal Dependency Relations*. URL: https://universaldependencies.org/u/dep/ (visited on 09/17/2020).

[31] Sebastian Weigelt and Walter F. Tichy. “Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 2. ISSN: 1558-1225. May 2015, pp. 819–820. DOI: 10.1109/ICSE.2015.264.

A. Appendix

A.1. Implementation

Type	Name	Description
String	documentation_Path	relative path to the textual documentation. The file should be a .txt file.
String	testDocumentation_Path	optional relative path to a textual example.
String	fileForInput_Path	relative path to write which is used to write the read in input into.
String	fileForResults_Path	relative path to a file in which the results are written. If the file is not existing it will be created.

Table A.1.: The general configurations that can be set in the configuration file.

Type	Name	Description
List<String>	ModelConnectionAgent _Analyzers	The list of analyzer types that should run on the connection state.
List<String>	ModelConnectionAgent _Solvers	The list of solver types that should run on the connection state.
double	RelationConnectionSolver _Probability	The probability for the identified link in the relation connection solver.
double	InstanceConnectionSolver _Probability	The probability for the identified link in the instance mapping connection solver.
double	InstanceConnectionSolver _ProbabilityWithoutType	The probability of the instance mapping connection solver, if the connection does not include the comparison of a type.

Table A.2.: The configurations that can be set for the connection generator.

Type	Name	Description
List<String>	separators _ToContain	the separators that are used for the separation features of the approach. The separators have to be in a form that they can be used with the contains functionality in Java.
List<String>	separators _ToSplit	the speparators that in a form that the split function of string can work with it. Therefore, some separators need escape sequences.
int	areWordsSimilar _MinLength	minimal length of word similarity for methods in SimilarityUtils.
int	areWordsSimilar _MaxLDist	maximal Levensthein distance for two words to be similar for methods in SimilarityUtils.
double	areWordsSimilar _DefaultThreshold	default threshold for similarity of two words of SimilarityUtils.
double	areWordsOfListsSimilar _WordSimilarityThreshold	threshold for the similarity of two words in the similarity function of two lists in SimilarityUtils.
double	areWordsOfListsSimilar _DefaultThreshold	default threshold for the similarity of two lists in SimilarityUtils.
double	getMostRecommendedIByRef _MinProportion	minimal proportion of two lists that need to be similar that both are handled as similar. Used in SimilarityUtils.
double	getMostRecommendedIByRef _Increase	increase for the method getMostRecommendedInstancesByReference in SimilarityUtils.
double	getMostLikelyMpByReference _Threshold	start threshold for the method getMostLikelyMappingByReference in SimilarityUtils.
double	getMostLikelyMpBReference _Increase	increase for the method getMostLikelyMappingByReference in SimilarityUtils.

Table A.3.: The configurations for helper classes, such as *SimilarityUtils*

Type	Name	Description
List<String>	TextExtractionAgent _Analyzers	the list of text extraction agent analyzers that should run, given as the name of their enums and separated by spaces.
List<String>	TextExtractionAgent _Solvers	the list of text extraction agent solvers that should run, given as the name of their enum and separated by spaces.
double	TextExtractionState _AddWithSeparators-AddProbability	probability that the mapping kind can be identified correctly in the out dep arcs analyzer.
double	SeparatedNamesAnalyzer _Probability	probability that the mapping kind can be identified correctly in the separated names analyzer.
double	OutDepArcsAnalyzer _Probability	probability that the mapping kind can be identified correctly in the out dep arcs analyzer.
double	NounAnalyzer _Probability	probability that the mapping kind can be identified correctly in the noun analyzer.
double	MultiplePartSolver _Probability	probability that the mapping kind can be identified correctly in the multiple part solver.
double	InDepArcsAnalyzer _Probability	probability that the mapping kind can be identified correctly in the in dep arcs analyzer.
double	ArticleTypeNameAnalyzer _Probability	probability that the mapping kind can be identified correctly in the article type name analyzer.
int	ExtractionState _MinTypeParts	minimal amount of parts of the type that the type is splitted and can be dentified by parts.

Table A.4.: The configurations of all parts of the text extraction level.

Type	Name	Description
List<String>	RecommendationAgent _Analyzers	the list of analyzer types that should run on the recommendation state.
List<String>	RecommendationAgent_Solvers	the list of solver types that should run on the recommendation state.
double	ExtractionDependentOccurrence-Analyzer_Probability	probability for the correct connection of name and type of an identified recommended instance in the extraction dependent occurrence analyzer.
double	NameTypeAnalyzer _Probability	probability for the correct connection of name and type of an identified recommended instance in the name type analyzer
double	ExtractedTermsAnalyzer _ProbabilityAdjacentNoun	probability for the correct connection of name and type of an identified recommended instance, when one of them is a term and the other is an adjacent noun mapping.
double	ExtractedTermsAnalyzer _ProbabilityJustName	probability for terms with no adjacent nouns and therefore without type, to be recommended.
double	ExtractedTermsAnalyzer _ProbabilityAdjacentTerm	probability for the correct connection of name and type of an identified recommended instance, when both are terms.
double	SeparatedRelationsSolver _Probability	probability for the correct connection of name and type of an identified recommended instance in the separated relations solver.
double	ReferenceSolver _Probability	probability for the correct connection of name and type of an identified recommended instance in the reference solver.
double	ReferenceSolver _ProportionalDecrease	decrease of the probability in the reference solver.
double	ReferenceSolver _AreNamesSimilarThreshold	threshold for words similarities in the reference solver.

Table A.5.: The configurations that can be set for the recommendation generator.

A.2. Evaluation

Hinweis: In dieser Evaluation geht es darum Modellelemente in Texten zu erkennen. Unter Modellelement werden Instanzen und Relationen zusammengefasst. Eine Instanz besteht aus einem Namen und einem Typen. Eine Relation besteht aus einer semantischen Bezeichnung (z.b. „ist ein“, oder „hat“), sowie mindestens zwei Instanzen.

Aufgabe 1: Markieren Sie alle Namen und Typen von Instanzen

a) Markieren Sie alle Namen und Typen von Instanzen im Text über TEAMMATES. Stellen Sie sich hierbei vor, Sie würden ein Architekturmodell zeichnen. Sie haben für die Markierungen drei Farben zur Verfügung, um Ihre Zustimmung auszudrücken:

Blau-Grün = Ich stimme werder zu noch lehne ab, dass dieser Name / Typ im Modell vorkommt.
DunkelGrün = Ich stimme zu, dass dieser Name/ Typ im Modell vorkommt.
HellGrün = Ich stimme voll und ganz zu, dass dieser Name/ Typ im Modell vorkommt.

b) Tragen Sie bitte alle Namen in der gegebenen Tablle ein. Fassen Sie hierbei Namen zusammen, die Ihrer Meinung nach zu einer Instanz gehören. Falls Ihrer Meinung nach zwei Namen unterschiedliche Instanzen kennzeichnen tragen Sie sie bitte in zwei verschiedenen Zeilen ein. Falls derselbe Namen mehrere unterschiedliche Instanzen referenziert, tragen Sie ihn bitte für jede referenzierte Instanz ein.

c) Falls möglich ergänzen Sie die identifizierte Instanz anschließend durch zugehörige Typen. Falls Sie keinen zugehörigen Typen zu einem Namen finden, lassen Sie das Typ-Feld frei. Falls Sie mehere Typen identifizieren, die semantisch auf den gleichen Typen referenzieren schreiben Sie diese in dasselbe Typ-Feld. Falls Sie mehrere Typen identifizieren, die Sie nicht als gleich betrachten würden, tragen Sie den/ die Namen mit den jeweiligen Typen in einzelne Zeilen.

d) Geben Sie nun Ihre Konfidenz für die Kombination aus Name und Typ an. Falls ein Feld von Name oder Typ frei ist geben Sie keine Konfidenz an. Beachten Sie: Die Konfidenz ist unabhängig davon, ob Sie diese Instanz im Modell zeichnen würden!

Beispiel:

Im folgenden wird Ihnen ein Beispielsatz dargestellt:
Text: „Die Klasse Keks enthält Schokoladenstücke. Die Klasse Schokostücke liegt im Modul Süßkram.“

In Aufgabenteil a) stimmt der Annotator in diesem Beispiel voll und ganz zu, dass „Klasse“ und „Süßkram“ in dem Modell vorkommen sollten. Bei „Keks“ und „Schokoladenstücke“ stimmt er zu, wohingegen er bei „Modul“ weder zustimmt noch ablehnt. Dies ist durch die Farbcodierung aus dem Text ablesbar.

In Aufgabenteil b) bestimmt der Studienteilnehmer alle Namen und listet diese auf. Falls seiner Meinung nach mehrere Namen die gleiche Instanz referenzieren, werden sie in der gleichen Zeile gelistet. Durch die Auslegung der einzelnen Teilnehmer kann es bereits hier zu unterschiedlichen Lösungsmöglichkeiten kommen.

In Aufgabenteil c) ergänzen beide Studienteilnehmer aus dem Beispiel die von ihnen durch Namen identifizierten Instanzen um Typen. Da in dem Text keine genauere Angaben zu dem Typen von „Schokoladenstücke“ vorliegen lässt Teilnehmer 2 den Typ dieser möglichen Instanz frei. Teilnehmer 1 hat jedoch „Schokoladenstücke“ und „Schokostücke“ einer Instanz zugeordnet. Da er zu „Schokostücke“ den Typen „Klasse“ zuordnet, wird dieser Typ für die gesamte Insanz eingetragen.

In Aufgabenteil d) kreuzen beide Teilnehmer graduell Ihre Zustimmung für die Kombination aus Name und Typ an. Während Teilnehmer 1 voll und ganz zustimmt, dass es eine Keks Klasse im Modell geben sollte stimmt Teilnehmer 2 dem zu. Beachten Sie: Die Wahrscheinlichkeit von einer Kombination aus Name und Typ ist unabhängig von der gewählten Farbcodierung aus dem Text.

Mögliche Lösung (Teilnehmer 1):

	Namen	**Typen**	**Stimme werder zu noch lehne ab**	**Stimme zu**	**Stimme voll und ganz zu**
Instanz 1	Keks	Klasse			x
Instanz 2	Schokoladenstücke, Schokostücke	Klasse		x	
Instanz 3	Süßkram	Modul			x

Mögliche andere Lösung (Teilnehmer 2):

	Namen	**Typen**	**Stimme werder zu noch lehne ab**	**Stimme zu**	**Stimme voll und ganz zu**
Instanz 1	Keks	Klasse		x	
Instanz 2	Schokoladenstücke				
Instanz 3	Schokostücke	Klasse		x	
Instanz 4	Süßkram	Modul		x	

Aufgabe 2: Eintragen aller Relationsbezeichner und -teilnehmer

Tragen Sie alle Relationen in die Tabelle ein, die sie im Text zwischen von Ihnen identifizierten Instanzen finden können. Tragen Sie für jede Relation einen semantischen Namen und alle Relationsteilnehmer ein. Fügen Sie bei jedem Auftreten der Relation in dieser Konstellation die Satznummer hinzu. Falls eine Relation mit unterschiedlichen Instanzen auftritt, listen Sie sie bitte extra auf. Geben Sie anschließen an, wie sehr Sie der Relation zustimmen.

Beispiel:

Text: „Die Klasse Keks enthält Schokoladenstücke. Die Klasse Schokostücke liegt im Modul Süßkram.“
Mögliche Lösung:

	Semantischer Name	**Verbundene Instanzen**	**Vorkommen (in Satznummern)**	**Stimme weder zu noch lehne ab**	**Stimme zu**	**Stimme voll und ganz zu**
Relation 1	enthält	A, C	1			x
Relation 2	liegt in	C, B	2			x

Aufgabe 3: Markieren Sie alle Namen und Typen von Instanzen, die im Modell vorkommen

Markieren Sie alle Namen und Typen von Instanzen im Text über TEAMMATES, die auch im Modell vorkommen. Gehen Sie dabei vor wie in Aufgabe 1. Führen Sie die Aufgabenteile nacheinander aus. Tragen Sie zusätzlich die Nummer der Modellinstanz, sowie die Satznummer in der Tabelle ein.

Instanz no.	Namen	Typen	Vorkommen (in Satznummern)	Stimme weder zu noch lehne ab	Stimme zu	Stimme voll und ganz zu

Aufgabe 4: Eintragen aller Relationsbezeichner und – teilnehmer, die im Modell vorkommen

Tragen Sie alle Relationen in die Tabelle ein, die im Modell vorkommen. Gehen Sie dabei vor wie in Aufgabe 2. Führen Sie die Aufgabenteile nacheinander aus. Tragen Sie zusätzlich die Nummer der Relation in der Tabelle ein.

Relation no.	Semantischer Name	Verbundene Instanzen	Vorkommen (in Satznummern)	Stimme weder zu noch lehne ab	Stimme zu	Stimme voll und ganz zu