Karlsruher Institut für Technologie

Software Design and Quality

Traceability Link Recovery und Inkonsistenzerkennung zwischen Modellen und informellen Diagrammen mithilfe struktureller Eigenschaften

Bachelorarbeit von

Jean Patrick Mathes

An der KIT-Fakultät für Informatik
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

1. Prüfer/Prüferin:	Prof. Dr.-Ing. Anne Koziolek
2. Prüfer/Prüferin:	Prof. Dr. Ralf H. Reussner
1. Betreuer/Betreuerin:	Dominik Fuchß, M.Sc.
2. Betreuer/Betreuerin:	Sophie Corallo, M.Sc.

10. Juli 2023 – 10. November 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 10.11.2023

. . . . . .

(Jean Patrick Mathes)

Zusammenfassung

Informelle Diagramme werden in der Softwareentwicklung zur Darstellung von Code und Architektur genutzt. Um erstellte Diagramme auch langfristig als Teil der Dokumentation erhalten zu können, müssen diese konsistent mit den dargestellten Inhalten gehalten werden. In dieser Arbeit wird ein Ansatz präsentiert, der automatisch Inkonsistenzen zwischen informellen Diagrammen in Boxen-und-Linien-Form und Architektur- sowie Codemodell findet. Im ersten Schritt wird bestimmt, welches der Modelle im Diagramm dargestellt wird. Dazu wird für jede Box im Diagramm nach allen entsprechenden Vorkommen in den Modellen gesucht und so eine Übereinstimmung zwischen dem Diagramm und den Modellen ermittelt. Dann werden das Diagramm und das Modell mit höherer Übereinstimmung für den zweiten Schritt in Graphen transformiert. Mit *graph matching* werden Nachverfolgbarkeitsverbindungen zwischen Diagramm und Modell bestimmt. Diese Verbindungen werden im dritten Schritt verwendet, um Inkonsistenzen zu finden. Zur Überprüfung und Evaluation werden sowohl synthetische als auch reale Diagramme verwendet. Auf den in der Evaluation verwendeten sechs Diagrammen wird ein durchschnittlicher F1-Score von 77,5% erzielt. Auch Inkonsistenzen, welche laut Goldstandard unerwartete sind, können dabei hilfreich sein.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Diagramme sind ein sinnvoller Bestandteil von Architekturdokumentation und Codebeschreibung. Sie helfen, Struktur und Zusammenhänge greifbar zu machen.

Eine Analyse von Baltes und Diehl [4] betrachtet Entwicklerverhalten bezüglich Skizzen und Diagrammen. Die Analyse zeigt, dass Diagramme in der Softwareentwicklung ein verbreitetes Werkzeug sind. Diese Diagramme sind größtenteils informell, sie folgen also keiner Spezifikation wie UML [1], sind aber häufig an solche Spezifikation angelehnt. Werden diese Diagramme erhalten, so dienen sie unter anderem als Dokumentation. Doch ein Teil der Diagramme wird nicht erhalten, sondern nach Erstellung schnell wieder verworfen. Verschiedene mögliche Gründe werden in der Analyse genannt, wobei zwei dieser Gründe zusammenhängen und hervorzuheben sind: Der erste Grund ist, dass Diagramme in Code überführt werden, die Diagramme dann also als redundant betrachtet werden. Der zweite Grund ist, dass die in den Diagrammen dargestellten Inhalte durch Änderungen am Code als überholt betrachtet werden. Beide Gründe deuten darauf hin, dass der Aufwand, die Diagramme konsistent zu halten, von den befragten Entwicklern als zu hoch angesehen wird. Die Lebenszeit und der Nutzen von Diagrammen könnten also erhöht werden, wenn es einfacher ist, die Konsistenz von Diagrammen und dem darin dargestellten Inhalten sicherzustellen.

Gemäß einer Analyse [8] von Open-Source-Projekten verwenden 40% der Projekte mit Architekturdokumentation in ihrer Dokumentation Diagramme. Gleichzeitig beinhalten nur wenige Open-Source-Projekte überhaupt eine Architekturdokumentation. Auch diese Beobachtung lässt sich auf den Aufwand von Softwaredokumentation schließen. Bei Änderungen der Architektur eines Projekts werden die beschreibenden Diagramme nicht automatisch konsistent gehalten. Es ist nicht auszuschließen, dass während der Entwicklung erstellte Diagramme nach einer Änderung gelöscht und die darin enthaltenen Informationen verloren gehen, da die Entwickler keine inkorrekten Diagramme in ihrer Dokumentation abbilden möchten.

Hier kann der in dieser Arbeit vorgestellte Ansatz helfen, der automatisch Diagramme mit Repräsentationen der darin abgebildeten Inhalte, also Code- und Architekturmodellen, abgleicht und sich entsprechende Elemente miteinander verbindet. Auf Basis dieser Verbindungen können dann Inkonsistenzen gesucht werden. Dabei muss die informelle Natur der Diagramme berücksichtigt werden. Dazu werden nur wenige, allgemeine Anforderungen an das Format des Diagramms und die darin vorkommenden Elemente gestellt. Dem liegt die Idee des Boxen-und-Linien-Diagramms zugrunde, welches nur beschriftete Objekte

(Boxen) und die Verbindungen dieser Elemente (Linien) enthält. Ein solches informelles Diagramm ist in Abbildung 1.1 dargestellt. Es ist auf Basis eines Diagramms aus der Softwaredokumentation von BigBlueButton [6] erstellt.

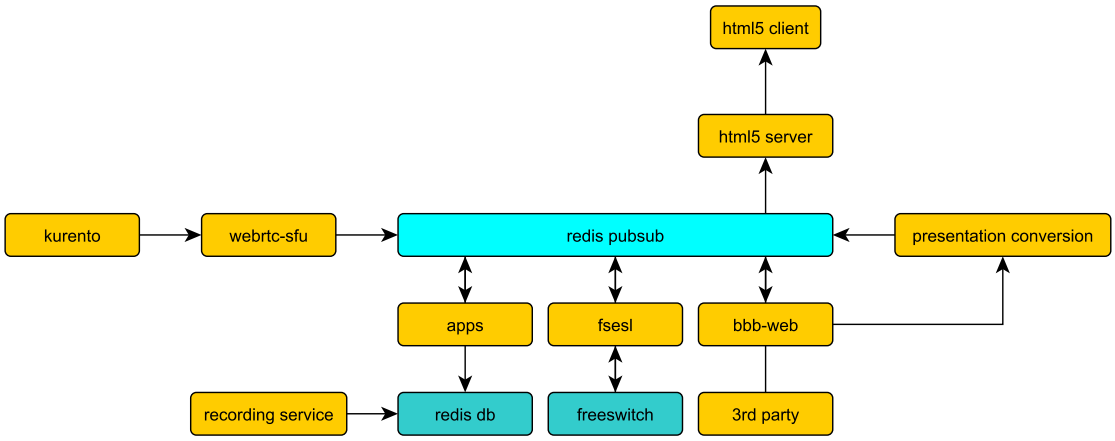

Abbildung 1.1: Architekturübersicht von BigBlueButton [6]

Im Folgenden wird die Entwicklung eines solchen Ansatzes zur automatisierten Verbindung von Diagramm und Modell untersucht. Dazu wird mit einem Blick auf notwendige Definitionen und Grundlagen begonnen. Darauf folgt ein Überblick über verwandte Arbeiten im Bereich der Nachverfolgbarkeitsverbindungen. Da die verwendete Boxen-und-Linien-Form als Graphen aufgefasst werden kann, werden dabei *graph matching*-Algorithmen betrachtet. Das darauffolgende Kapitel beinhaltet eine Beschreibung der für die Entwicklung des Ansatzes betrachteten Diagramme. Dann wird der Aufbau und die Entwicklung des Ansatzes dargestellt. Zuletzt wird der Ansatz evaluiert.

2 Grundlagen

In diesem Abschnitt werden die notwendigen Grundlagen beschrieben und Begriffe definiert. Zuerst wird der Bereich der *trace link recovery* beleuchtet, da diese Arbeit sich mit einem Teilbereich dessen befasst. Darauf folgt eine Beschreibung von Textähnlichkeitsfunktionen, einem wichtigen Baustein dieser Arbeit. Dann werden die beiden zu verbindenden Seiten, also Diagramme und Modelle, betrachtet. Als Nächstes wird der Begriff der Inkonsistenz und Arten von Inkonsistenzen definiert. Darauf folgend wird der in der Arbeit verwendete Algorithmus Similarity Flooding beschrieben, dieser wird für *graph matching* eingesetzt. Zuletzt folgt eine Definition der Graphrepräsentation von Diagrammen und Modellen.

Metriken In dieser Arbeit werden verschiedene Metriken verwendet, um Ergebnisse miteinander zu vergleichen. Sowohl die tatsächlich vom Ansatz erstellten Ergebnisse als auch die erwarteten Ergebnisse sind eine Menge von Angaben. Angaben, welche in beiden Mengen existieren, sind korrekte Angaben. Angaben, welche nur in den tatsächlichen Ergebnissen sind, gelten als unerwartet. Entsprechend sind Angaben, welche nur in den erwarteten Ergebnissen existieren, fehlend. Für diese Arbeit sind die wichtigsten Metriken Ausbeute, Präzision und der F1-Score. Ein Überblick über die verwendeten und weitere Metriken ist von Hicks u. a. [14] bereitgestellt, die relevanten Informationen werden im Folgenden dargestellt. Ausbeute (*recall*) ist die Anzahl der korrekten Angaben, geteilt durch die Anzahl aller Angaben in dem erwarteten Ergebnis. Präzision (*precision*) ist die Anzahl der korrekten Angaben, geteilt durch die Anzahl aller Angaben in dem tatsächlichen Ergebnis. Der F1-Score ist das harmonische Mittel von Präzision und Ausbeute, also:

$$2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.1}$$

2.1 Nachverfolgbarkeitsverbindungen

In Traceability Fundamentals [10] werden die folgenden Begrifflichkeiten definiert, die auch für die Beschreibung des vorgestellten Ansatzes verwendet werden. Eine Nachverfolgbarkeitsverbindung (*trace link*) ist eine Assoziation zweier Elemente (*trace artifact*). Die Verbindung kann dabei zusätzliche Daten wie Attribute beinhalten. Elemente können alle Bestandteile eines Software-Projekts sein, zum Beispiel Klassen, Anforderungen oder Dateien. Der Begriff Nachverfolgung (*trace*) bezeichnet die Kombination dreier Bestandteile (*trace element*): Das erste Element, das zweite Element und die Verbindung dieser.

Explizite Verbindungen zwischen Elementen müssen erst erstellt werden. Dies kann während der Erstellung der Elemente selbst geschehen, dies wird als *trace capture* bezeichnet. Alternativ können die Verbindungen im Nachhinein erstellt werden, dann wird von *trace recovery* gesprochen. Der vorgestellte Ansatz führt die Erstellung der Verbindungen im Nachhinein durch, es handelt sich also um *trace recovery*.

Das Werkzeug ArDoCo [3] befasst sich mit Nachverfolgbarkeitsverbindungen. ArDoCo steht für Architecture Documentation Consistency, es sucht also nach Nachverfolgbarkeitsverbindungen zwischen Dokumentation in natürlicher Sprache und Architekturmodell. Gleichzeitig ist ArDoCo auch ein Framework, das für verschiedene Nachverfolgungsaufgaben verwendet werden kann. Dazu nutzt ArDoCo eine Pipeline-Architektur, in der eine Menge von aufeinanderfolgenden Schritten ausgeführt wird. Es können neue Agenten erstellt werden, diese sind Schritte, welche selbst aus als Informanten bezeichneten Schritten bestehen. ArDoCo beinhaltet bereits viele Agenten und Informanten und wird durch diese Arbeit um eine fest definierte Pipeline sowie alle darin enthaltenen Agenten erweitert.

2.2 Textähnlichkeitsfunktionen

Vergleichen von Texten ist für diese Arbeit von großer Bedeutung. Insbesondere ein Maß der Ähnlichkeit zweier Texte wird benötigt. Für die folgende Arbeit wird definiert, dass eine Funktion f eine Textähnlichkeitsfunktion ist, wenn gilt:

1. f akzeptiert als Eingabe zwei Texte beliebiger Länge
2. f berechnet einen Wert im Bereich $[0; 1]$, dieser Wert wird als Ähnlichkeit bezeichnet
3. Je ähnlicher die Texte desto größer der Wert. Sind die Texte identisch, ist die Ähnlichkeit 1

Gomaa und Fahmy [9] beschreiben in ihrer Übersicht verschiedene Verfahren zur Berechnung der Ähnlichkeit von Texten, von denen eine Auswahl in dieser Arbeit verwendet wird.

Die Levenshtein-Ähnlichkeit verwendet die von der Levenshtein-Distanz berechnete Änderungsdistanz zweier Texte. Die Levenshtein-Distanz gibt die minimale Anzahl an nötigen Operationen auf Zeichen an, um einen Text in den anderen zu überführen. Dabei erlaubte Operationen sind Einfügen, Ersetzen und Löschen von Zeichen. [12] Die Distanz wird durch die Länge des längeren der beiden Texte geteilt, denn die Länge ist die maximale Distanz die berechnet werden kann. Dieser Wert ist die relative Distanz zweier Texte, zieht man von diesem Wert 1 ab, so erhält man die Levenshtein-Ähnlichkeit.

Die Jaro-Winkler-Ähnlichkeit basiert auf einer erweiterten Form der Jaro-Distanz. Für die Jaro-Distanz werden die Anzahl von übereinstimmenden Zeichen sowie die Anzahl von Vertauschungen genutzt. Die Jaro-Winkler-Ähnlichkeit erweitert die Jaro-Distanz so, dass Übereinstimmung am Wortanfang stärker gewichtet wird. [29]

Sowohl Levenshtein- als auch Jaro-Winkler-Ähnlichkeit arbeiten primär auf einzelnen Zeichen. Im Gegensatz dazu arbeitet die Jaccard-Ähnlichkeit auf Token, also beliebigen Einheiten wie Silben, Wörtern oder auch Zeichen. Es wird die Anzahl gemeinsamer Token im Verhältnis zu einzigartigen Token berechnet [9]. Im einfachsten Fall bedeutet dies, dass beide Texte als Menge ihrer Zeichen betrachtet werden. Dann ist die Jaccard-Ähnlichkeit die Größe des Schnittes beider Texte, geteilt durch die Größe der Vereinigung.

2.3 Diagramme

Diagramme sind im Kontext des vorgestellten Ansatzes eine Menge von Boxen und Linien. Boxen können beschriftet sein. Jede Box beinhaltet eine möglicherweise leere Menge von anderen Boxen. Jede Box hat eine Menge von Linien, die von ihr ausgehen, diese kann auch leer sein. Linien haben eine Richtung und verbinden zwei Boxen. Abbildung 2.1 zeigt, wie ein Diagramm als Datenstruktur abgebildet werden kann. Für alle folgenden Beschreibungen werden Beinhaltung und Linien als Eigenschaften der beinhaltenden Box beziehungsweise der Ausgangsbox betrachtet. Im Kontext der vorhergegangenen Definition sind in Diagrammen nur Boxen als Elemente (*trace element*) zu betrachten.

Interpretation anderer Formen Alle Elemente in einem Diagramm, also Rechtecke, Kreise, Symbole und weitere, werden als Box interpretiert. Alle Verbindungen dazwischen, also Pfeile, Kurven und Striche, werden als Linien interpretiert. Eine ungerichtete Verbindung wird als zwei gegenläufige Linien interpretiert. Sobald eine Box teilweise mit einer größeren Box überlappt, ist sie in der größeren Box beinhaltet. Die verwendete Form stellt minimale Anforderungen an die Darstellung von Diagrammen. Dadurch wird ermöglicht, die Skizzen, informelle Diagramme, einer Spezifikation folgenden Diagramme und die Ausgabe von Bilderkennung, auf diese Form zu reduzieren. Automatische Konvertierung zu der Boxen-und-Linien-Form oder Bilderkennung wird in dieser Arbeit explizit nicht behandelt.

Informelle Diagramme Insbesondere informelle Diagramme stehen im Fokus dieser Arbeit. Ein informelles Diagramm ist in diesem Kontext ein Diagramm, welches keiner Spezifikation wie zum Beispiel UML [1] folgt. Es gibt also kein definiertes Metamodell, welches die Semantik des Diagramms angibt, sondern nur Konventionen. Informelle Diagramme können aber dennoch Elemente des UML-Standards nutzen. Auch kann es sinnvoll sein, die Namen der Diagrammtypen in UML für informelle Diagramme zu nutzen.

Die Art der Diagrammerstellung ist für den Ansatz nicht relevant, sowohl Skizzen auf Papier, digitale Zeichnungen als auch in Anwendungen für Graphen erstellte Darstellungen sind gültig, sofern das Diagramm in Boxen-und-Linien-Form überführt ist. Wird das Diagramm mit Tools für Modellierung erstellt, zum Beispiel in einem UML-Editor, so ist der Ansatz immer noch darauf anwendbar. In diesem Fall ist es aber sinnvoller, spezielle Werkzeuge zu nutzen, welche die formale Spezifikation, der das Diagramm folgt, ausnutzen.

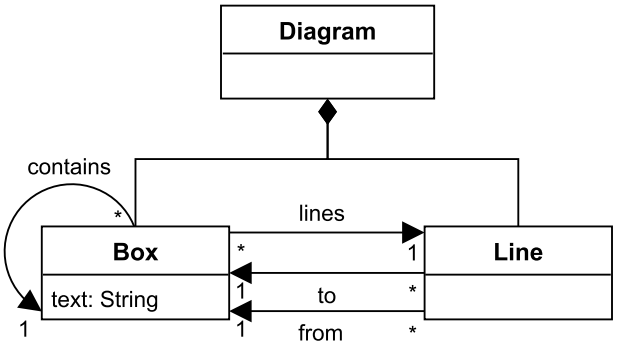

Abbildung 2.1: Diagramm-Struktur als Klassendiagramm

Diese Darstellung von Diagrammen ist in ähnlicher Form bereits eingeführt und formalisiert worden. Malton und Holt [20] definieren das *Nested Boxes and Arrows model*, als NBA-Modell bezeichnet. Das NBA-Modell ist im Gegensatz zu der hier verwendeten Darstellung formell. So werden über Attribute jedem Pfeil eine klare Bedeutung zugeordnet. Dies ist möglich, da das NBA-Modell mehr Informationen als die hier verwendete Darstellung beinhalten kann. Es ist erlaubt, Boxen und Linien zu typisieren und Attribute an Boxen und Linien anzuhängen. Malton und Holt [20] formalisieren das NBA-Modell als Graphen mit bestimmten Eigenschaften. Auch die hier verwendete Darstellung ist graph-ähnlich, eine Transformation von und zu Graphen ist möglich.

Stellt ein Diagramm Teile eines Architekturmodells dar, so wird es im Folgenden als Architekturdiagramm bezeichnet. Stellt ein Diagramm Teile eines Codemodells dar, so wird es im Folgenden als Codediagramm bezeichnet.

2.4 Modelle

Um den Quelltext und die Architektur eines Projekts zu repräsentieren, werden Codemodelle und Architekturmodelle verwendet. Ein Modelltyp wird dabei durch ein Metamodell beschrieben. Für beide Modelltypen werden, bis auf eine Anpassung im Codemodell, anfänglich die gleichen Metamodelle wie in ArDoCo [3] genutzt. ArDoCo selbst baut auf der Arbeit von Keim u. a. [17] auf, diese verwendet nur Architekturmodelle. Von Telge [27] wird dann ein Codemodell zu ArDoCo hinzugefügt. Beide folgenden Abbildungen zeigen die Metamodelle in Form eines UML-Klassendiagramms [1].

2.4.1 Architekturmodell

Das Architekturmetamodell, dargestellt in Abbildung 2.2, besteht aus Architekturelementen (*ArchitectureItem*). Diese sind Entitäten mit Namen. Architekturelemente haben drei Unterklassen; Komponenten (*Component*) beinhalten beliebig viele Unterkomponenten.

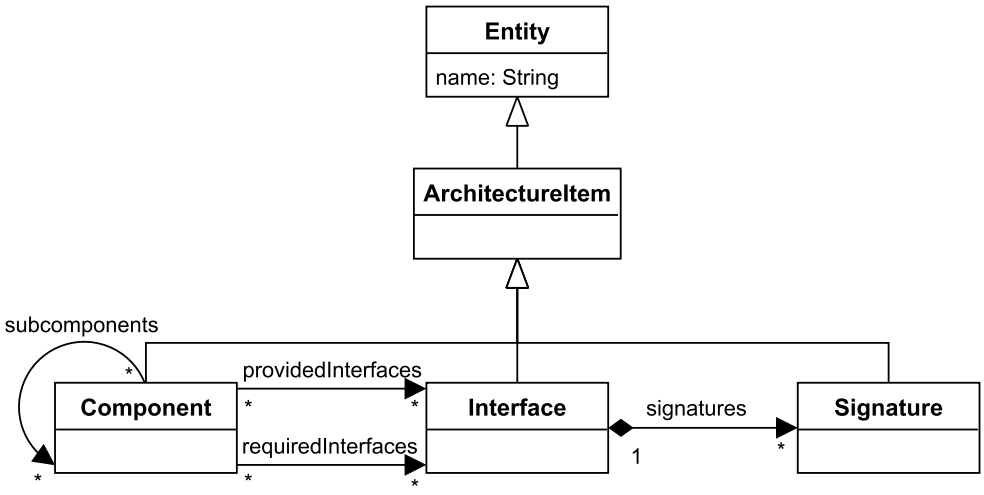

Abbildung 2.2: Das Architektur-Metamodell [27, S. 28]

Jede Komponente benötigt eine Menge von Schnittstellen und stellt eine Menge dieser zur Verfügung. Schnittstellen (*Interface*) bestehen aus einer beliebigen Menge von Signaturen (*Signature*). Für den vorgestellten Ansatz sind Komponenten und Schnittstellen sowie deren Beziehungen relevant. Im Kontext der vorhergegangenen Definition sind Klassen, Schnittstellen und Pakete als Elemente (*trace element*) zu betrachten.

2.4.2 Codemodell

Das Code-Metamodell, siehe Abbildung 2.3, besteht aus Quelltextelementen (*CodeItem*). Diese sind alle Entitäten (*Entity*) und haben einen Namen. Es gibt drei direkte Unterklassen der Quelltextelemente:

- Module (*Module*) beinhalten beliebig viele Quelltextelemente. Sie sind entweder Pakete (*Package*), Übersetzungseinheiten (*CompilationUnit*) oder kombinierte Einheiten (*CodeAssembly*). Da Übersetzungseinheiten einzelne Dateien darstellen, haben diese einen Pfad und sind einer Programmiersprache zugeordnet.

- Berechnungsobjekte (*ComputationalObject*) haben nur eine Unterklasse, die Kontrollelemente (*ControlElement*). Diese stellen aufrufbare Funktionen und Methoden dar.

- Datentypen (*Datatype*) haben beliebig viele Implementierungs- und Erweiterungsbeziehungen. Zusätzlich kann ein Datentypen einen Eltern-Datentypen haben. Das ist zum Beispiel der Fall, wenn eine Klasse innerhalb einer anderen Klasse erstellt ist. Diese Beziehung ist im originalen Codemodell von Telge [27] nicht vorhanden. Die

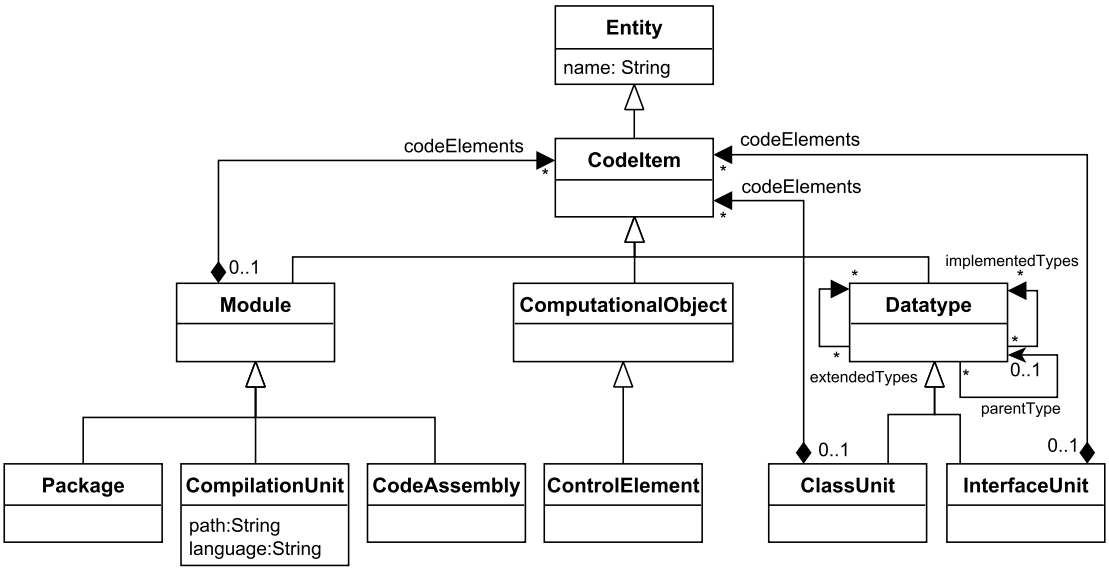

Abbildung 2.3: Das Code-Metamodell [27, S. 26], um *parentType* ergänzt

zwei konkreten Datentypen sind Klassen (*ClassUnit*) und Schnittstellen (*InterfaceUnit*). Beide bestehen wieder aus beliebig vielen Quelltextelementen, beinhalten in den verwendeten Modellen aber nur Berechnungsobjekte.

Für den vorgestellten Ansatz sind insbesondere Pakete, Klassen und Schnittstellen sowie deren Beziehungen relevant. Im Kontext der vorhergegangenen Definition sind Komponenten und Schnittstellen als Elemente (*trace element*) zu betrachten.

2.5 Erweiterung des Codemodells um Abhängigkeiten

Dieser Abschnitt beschreibt eine Erweiterung des Codemodells und der verwendeten Graphen um Abhängigkeit zwischen Klassen und Paketen. Durch diese Erweiterung beinhaltet das Modell mehr Informationen über die Struktur des Codes. Somit können Linien, welche Abhängigkeits- und Benutzt-Beziehung außerhalb von Vererbung und Implementierung darstellen, vom Ansatz berücksichtigt werden.

Die Änderung des Modells erweitert jeden Datentyp (*Datatype*) um eine Menge von referenzierten Datentypen. Ein Datentyp gilt als referenziert, wenn dieser im Quelltext einer Klasse direkt über den Bezeichner verwendet wird. Dazu gehören zum Beispiel Typen in Variablendeklarationen, Methodensignaturen und Attributsdeklarationen.

Für Pakete werden Abhängigkeiten auf Basis der Klassen-Abhängigkeiten erstellt. Ist eine Klasse abhängig von einer Klasse in einem anderen Paket, so ist das Paket abhängig von dem anderen Paket. Diese Abhängigkeiten werden nicht explizit im Modell dargestellt, da

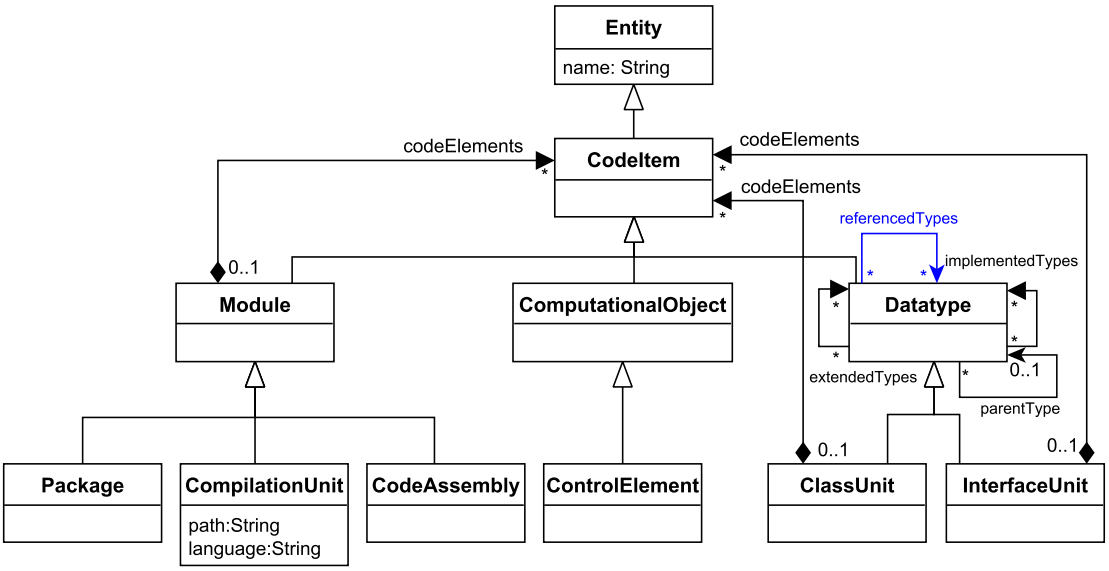

Abbildung 2.4: Das erweiterte Code-Metamodell, Änderungen zu Abbildung 2.3 in blau

diese bereits implizit vorhanden sind. Aus der Definition der Paket-Abhängigkeit folgt, dass Pakete auch Gegenseitig voneinander abhängig sein können. Die Abhängigkeiten zwischen Paketen werden als `default`-Kanten dargestellt. Mit dieser Regel ist es möglich, dass in realen Projekten eine große Anzahl von Paketen voneinander abhängig sind. Somit kann es bei der Inkonsistenz-Erkennung zu vielen als fehlend erkannten Kanten kommen. Da Inkonsistenzen aber nicht bedeuten, dass ein Diagramm fehlerhaft ist, sondern nur auf mögliche Fehler hinweisen, ist dies nicht problematisch.

2.6 Betrachtete Typen von Inkonsistenzen

Um mögliche Typen von Inkonsistenzen zu identifizieren, werden Änderungen an Code und Modellen betrachtet. Erst durch Änderungen entstehen Inkonsistenzen in vorher untereinander konsistenten Artefakten. Für das weitere Vorgehen werden *refactorings* auf Basis einer Analyse von Dig und Johnson [7] betrachtet. Darin werden *refactorings* definiert als Änderungen, welche nicht das Verhalten des Codes oder Modells ändern, sondern nur die Struktur. Die Analyse zeigt, dass in verschiedenen Projekte mehr als 80% der *breaking changes* durch *refactorings* verursacht wurden. Somit sind *refactorings* ein prominenter Bestandteil der Softwareentwicklung. Es ist also möglich, dass Inkonsistenzen durch *refactorings* entstehen, sofern diese nicht auf alle zusammenhängenden Artefakte gleich angewendet werden. Die Analyse beschreibt eine Menge von strukturellen Änderungen, welche an den Projekten durchgeführt werden. Angelehnt an diesen Änderungen ergibt sich die folgende Liste von in dieser Arbeit betrachteten Änderungsklassen:

move Ein Element wird in der Hierarchie verschoben. In einem Klassendiagramm ist dies zum Beispiel das Verschieben einer Klasse in ein anderes Paket. In einem allgemeinen Diagramm ist dies das Verschieben einer Box in eine andere Box.

rename Ein Element wird umbenannt.

create Ein neues Element wird erstellt.

delete Ein Element wird gelöscht.

connect Ein Element wird mit einem anderen in Beziehung gesetzt. Das kann zum Beispiel das Hinzufügen einer Vererbungsbeziehung oder einer neuen Abhängigkeit sein. Dies verändert nicht die Hierarchie der Elemente.

disconnect Eine Beziehung, in Diagrammen als Linie dargestellt, zwischen zwei Elementen wird gelöscht. Das kann zum Beispiel das Entfernen einer Abhängigkeit oder Vererbungsbeziehung sein. Dies verändert nicht die Hierarchie der Elemente.

Die Änderungen sind abstrahiert für die Anwendung auf Boxen-und-Linien-Diagramme, Codemodell und Architekturmodell. Die letzten beiden Änderungen werden so nicht direkt in der Analyse betrachtet, sondern ergeben sich aus der Anwendung von *create* und *delete* auf Beziehungen. Viele weitere mögliche *refactorings* in der Analyse befassen sich mit Methoden und Attributen, welche für den Ansatz nicht betrachtet werden.

Ausgehend von den Änderungsklassen werden im Folgenden Inkonsistenz-Typen definiert. Diese sind aus Sicht des Diagramms definiert.

- `hierarchy_inconsistency`: Diese Inkonsistenz kann unter anderem durch *move* entstehen. Sie gibt an, dass die Hierarchie der Boxen nicht mehr mit der Hierarchie der Elemente übereinstimmt.
- `name_inconsistency`: Diese Inkonsistenz kann durch *rename* entstehen. Sie gibt an, dass der Name eines Elements nicht mehr mit dem Namen der Box übereinstimmt.
- `missing_box`: Diese Inkonsistenz kann durch *create* und *delete* entstehen. Wird eine Box im Modell erstellt, aber nicht im Diagramm, so fehlt die Box im Diagramm. Wird eine Box im Diagramm gelöscht, aber nicht im Modell, so fehlt auch dann die Box im Diagramm.
- `unexpected_box`: Wie `missing_box` kann diese Inkonsistenz durch *create* und *delete* entstehen. Die Fälle sind hier entgegengesetzt.
- `missing_line`: Diese Inkonsistenz kann durch *connect* und *disconnect* entstehen. Wird eine Linie im Modell erstellt, aber nicht im Diagramm, so fehlt die Linie im Diagramm. Wird eine Linie im Diagramm gelöscht, aber nicht im Modell, so fehlt auch dann die Linie im Diagramm.
- `unexpected_line`: Wie `missing_line` kann diese Inkonsistenz durch *connect* und *disconnect* entstehen. Die Fälle sind hier entgegengesetzt.

2.7 Graph Matching

Der Similarity Flooding-Algorithmus von Melnik, Garcia-Molina und Rahm [21] ist für das *matching* von Modellen in Form von Graphen konzipiert. Die folgende Darstellung ist angepasst, um eine Implementierung zu erleichtern. Die Eingabe besteht aus zwei gerichteten und schleifenfreien Graphen $A = (V^A, E^A)$, $B = (V^B, E^B)$ mit typisierten Kanten, sowie einer initialen Zuordnung σ_0. Ausgehend von der initialen Zuordnung werden iterativ neue Zuordnungen berechnet. Die grundlegende Idee ist, dass die Nachbarn zweier ähnlicher Knoten auch ähnlich sind. Daher wird Ähnlichkeit von ähnlichen Knoten zu ihren Nachbarn propagiert. In Iteration i (mit $i \geq 0$) wird Zuordnung σ_i genutzt, um σ_{i+1} zu berechnen. Eine Zuordnung σ_i bestimmt für ein Paar (a, b) mit $a \in V^A$, $b \in V^B$ eine Ähnlichkeit $s \in [0; 1]$. Kanten sind ein Tupel (x, l, y) mit $x, y \in V^A$, l ist der Kantentyp, auch als *label* bezeichnet. Mehrere Kanten zwischen zwei Knoten sind erlaubt, sofern die Typen sich unterscheiden. Zwei Kanten in entgegengesetzte Richtungen können den gleichen Typen haben.

In einer erweiterten Fassung der beschreibenden Arbeit [22] wird eine formale Definition präsentiert. Diese ist direkt auf den beiden Graphen A und B definiert und nutzt die im Folgenden beschriebenen Datenstrukturen nur implizit.

2.7.1 Aufbau der benötigten Datenstrukturen

Der Algorithmus arbeitet auf einem kombinierten Graphen, dem *propagation graph* $PG(A, B)$. Dieser Graph verbindet alle betrachteten Kombinationen zweier Knoten aus A und B mit deren Nachbarn und gewichtet wie stark Ähnlichkeit zwischen ihnen übertragen wird.

Der *propagation graph* $PG(A, B)$ selbst baut auf dem *pairwise connectivity graph* $PCG(A, B) = (V^{PCG}, E^{PCG})$ auf. Dieser Graph ist eine Kombination der beiden Eingabegraphen. Alle Knoten in $PCG(A, B)$ sind Paare von Knoten in den Graphen A und B. Nicht alle möglichen Paare $V^A \times V^B$ sind Teil von $PCG(A, b)$. Nur Paare $(x_A, x_B) \in V^{PCG}$, mit $x_A \in V^A$, $x_B \in V^B$, die beide Teil einer Kante mit gleichem Typen l sind, werden in PCG eingefügt. Es gilt für eine Kante $e = ((x_A, x_B), l, (y_A, y_B))$ die in Gleichung 2.2 beschriebene Eigenschaft.

$$e \in E^{PCG} \iff (x_A, l, y_A) \in E^A \wedge (x_B, l, y_B) \in E^B \tag{2.2}$$

Die Knotenmenge des *propagation graph* $PG(A, B)$ ist gleich der des Graphen $PCG(A, B)$. Zur Bestimmung der Kantenmenge wird jeder Knoten $x = (x_A, x_B)$ in $PG(A, B)$ betrachtet. Für alle aus x ausgehenden Kanten $(x, l, y) \in E^{PCG}$ wird eine Kante $(x, l, w^{out}(x, l), y)$ in $PG(A, B)$ eingefügt. Dabei ist $w^{out}(x, l)$ ein Gewicht, welches für alle ausgehenden Kanten des gleichen Typs l gleich ist.

Das Gewicht $w^{out}(x, l)$ kann auf verschiedene Arten berechnet werden. Eine Möglichkeit ist das Inverse der Anzahl der ausgehenden Kanten des Typs l, beschrieben in Gleichung 2.3.

$$w^{out}(x, l) = \frac{1}{|\{y \mid (x, l, y) \in E^{PCG}\}|} \tag{2.3}$$

Auf ähnliche Weise werden auch die eingehenden Kanten in $PCG(A, B)$ betrachtet und zu ausgehenden Kanten in $PG(A, B)$ umgewandelt. Hier wird für alle in x eingehenden Kanten $(y, l, x) \in E^{PCG}$ eine Kante $(x, l, w^{in}(x, l), y)$ in $PG(A, B)$ eingefügt. Die eingefügten Kanten gehen von x aus, somit ist es möglich, dass mehr als eine Kante mit Typ l von Knoten x zu einem Knoten y existiert. Es wird die gleiche Funktion wie für ausgehende Kanten verwendet, aber angepasst, um eingehende Kanten zu betrachten. Kanten mit Gewicht 0 und Knoten ohne Kanten kommen im *propagation graph* nicht vor, da diese keine Auswirkung auf die Ähnlichkeit haben.

Die allgemeine Formel für die Gewichte nutzt den Ausgangs- beziehungsweise Eingangsgrad von Knoten in A und B. Der Grad in A wird mit dem in B multipliziert, sodass das Gewicht nur größer als 0 ist, wenn beide Knoten mindestens eine Kante des Typs l haben. Zur Berechnung lässt sich eine allgemeine Formel aufstellen, mit welcher der Graph $PG(A, B)$ nicht nötig ist, um die Kantengewichte von $PG(A, B)$ zu berechnen. Die allgemeine Formel ist in Gleichung 2.4 dargestellt, dabei gibt $card_{\{in,out\}}(x, l, G)$ die Anzahl der Kanten des Typs l an, die in G in x eingehen (in) oder ausgehen (out). Die in Gleichung 2.4 dargestellte Formel wird von Melnik, Garcia-Molina und Rahm [21] als *inverse product* bezeichnet.

$$w(x, l) = \frac{1}{card_{\{in,out\}}(x, l, A) \cdot card_{\{in,out\}}(x, l, B)} \tag{2.4}$$

In Abbildung 2.5 sind die Graphen dargestellt, welche für Similarity Flooding verwendet werden. Die ersten beiden Graphen sind die Eingabegraphen A und B. Dann ist der *pairwise connectivity graph* $PCG(A, B)$ dargestellt. Zuletzt folgt der *propagation graph* $PG(A, B)$. Die Gewichte in $PG(A, B)$ sind anhand der Formel *inverse product* berechnet und in der Abbildung immer am Anfang der Kante dargestellt.

Eine Alternative zu *inverse product* ist *inverse average*, diese Funktion berechnet statt eines Produkts die Summe. Die Bedingung in Gleichung 2.2 gilt dann nicht mehr, statt der Konjunktion ist Disjunktion nötig, da es ausreicht, dass einer der Knoten in A oder B eine Kante des Typs l hat. In Gleichung 2.5 ist die allgemeine Formel für *inverse average* dargestellt. Dieser Formel wurde in Experimenten von Melnik, Garcia-Molina und Rahm [22] eine im Vergleich zu anderen Formeln bessere Genauigkeit nachgewiesen. Daher wird hier auch diese Formel verwendet.

$$w(x, l) = \frac{2}{card_{\{in,out\}}(x, l, A) + card_{\{in,out\}}(x, l, B)} \tag{2.5}$$

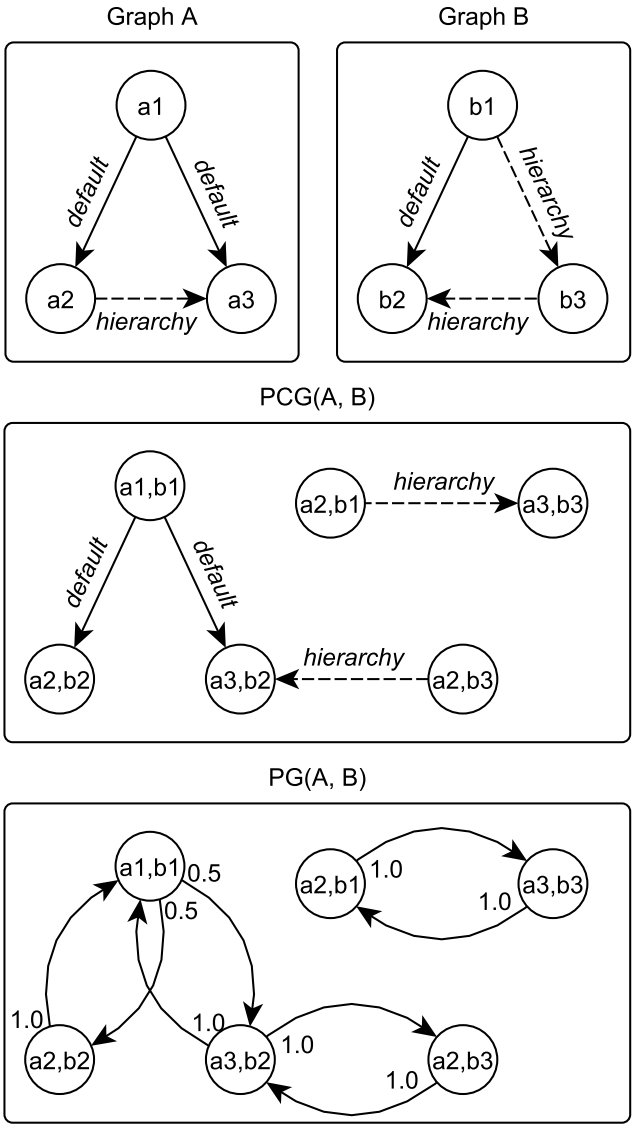

Abbildung 2.5: Die Graphen, welche vom Similarity Flooding-Algorithmus verwendet werden

2.7.2 Iterativer Ablauf des Algorithmus

Sobald der *propagation graph* $PG(A, B)$ erstellt ist, beginnt der iterative Teil. Hier wird die Funktion $flood(\sigma_i)$ verwendet, die folgendermaßen eine neue Zuordnung bestimmt: Für jeden Knoten $x \in V^{PG}$ werden alle eingehenden Kanten $in(x)$ betrachtet. Dabei wird entlang jeder dieser Kanten $(y, l, c, x) \in in(x)$ die Ähnlichkeit $\sigma_i(y)$ propagiert. Das Gewicht c wird dabei als Koeffizient verwendet. Somit verteilt jeder Paar-Knoten in

$PG(A, B)$ seine Ähnlichkeit auf alle Nachbarn. Es gilt die Definition in Gleichung 2.6.

$$flood(\sigma_i)(x) = \sum_{(y,l,c,x) \in in(x)} c \cdot \sigma_i(y) \tag{2.6}$$

Damit lässt sich die Fixpunktgleichung für den Algorithmus aufstellen, siehe dazu Gleichung 2.7. Diese wird im Original als *basic* bezeichnet.

$$\sigma_{i+1} = normalize(\sigma_i + flood(\sigma_i)) \tag{2.7}$$

Die Addition von Zuordnungen ist hier komponentenweise definiert. Die Zuordnung σ_{i+1} wird mit der Funktion *normalize* normalisiert, sodass alle Werte zwischen 0 und 1 liegen. Dazu wird jeder Wert durch den größten Wert geteilt. Alternative Fixpunktgleichungen sind in Abbildung 2.6 dargestellt, es werden zusätzlich zu der bereits bekannten Funktion *basic* auch die Gleichungen A, B und C definiert. In der Arbeit zu Similarity Flooding wird die Verwendung der Gleichung C empfohlen.

$$basic \quad \sigma_{i+1} = normalize(\sigma_i + flood(\sigma_i))$$

$$A \quad \sigma_{i+1} = normalize(\sigma_0 + flood(\sigma_i))$$

$$B \quad \sigma_{i+1} = normalize(flood(\sigma_0 + \sigma_i))$$

$$C \quad \sigma_{i+1} = normalize(\sigma_0 + \sigma_i + flood(\sigma_0 + \sigma_i))$$

Abbildung 2.6: Verschiedene Fixpunktgleichungen für Similarity Flooding

Die Iteration wird so lange ausgeführt, bis der Abstand zwischen zwei aufeinanderfolgenden Zuordnungen kleiner als ein Schwellwert ϵ ist. Dieser Parameter wird im Folgenden auch `epsilon` genannt. Um endlose Iterationen zu vermeiden, wird zusätzlich eine maximale Anzahl an Iterationen festgelegt. Dieser Parameter wird als `max_iterations` bezeichnet. Eine vereinfachte Darstellung von Similarity Flooding ist in Algorithmus 1 dargestellt.

Algorithmus 1 Similarity Flooding

Eingabe: σ_0, A, B
Ausgabe: σ_n mit $n \in [0, \texttt{max\_iterations}]$

```
pg ← PG(A, B)
δ ← ∞
i ← 0
while δ ≥ ε ∧ i ≤ max_iterations do
    σ_{i+1} ← normalize(σ_i + flood(σ_i))      ▷ Fixpunktgleichung basic
    δ ← ||σ_{i+1} − σ_i||
    i ← i + 1
end while
return σ_i
```

2.7.3 Verhalten des Algorithmus

Verschiedene Eigenschaften des Algorithmus machen die Wahl eines geeigneten Schwellwerts ϵ notwendig. Ein hoher Wert führt zu einer geringen Anzahl von Iterationen, was die Laufzeit reduziert. Ein niedriger Wert ermöglicht es den Ähnlichkeiten, sich mehr an ihr Ziel anzunähern, sofern die Werte konvergieren. Bei hoher Anzahl von Iterationen konvergieren die meisten Ähnlichkeiten zu einem Wert. Der Wert kann dabei im gesamten Bereich zwischen 0 und 1 liegen. Für einige Paare findet eine Konvergenz gegen 0 statt. Sollten diese Paare für die Anwendung eine sinnvolle Kombination sein, so ist ein niedriger Wert für ϵ in diesen Fällen nicht geeignet, denn dies erschwert die Unterscheidung zwischen verschiedenen Ähnlichkeiten sowie die Auswahl der Paare.

Auch die Auswahl der Fixpunktgleichung muss bedacht werden. Die Gleichung *basic* verwendet nicht die initiale Zuordnung σ_0. Dadurch ist ein Verlust der darin enthaltenen Information möglich. Andere Gleichungen nutzen die initiale Zuordnung. Sollten dann unerwünschte Informationen enthalten sein, so werden diese dann in jeder Iteration weiter propagiert. Der Parameter ϵ hat eine Auswirkung auf die Gewichtung der initialen Zuordnung.

2.8 Graphrepräsentation von Modellen und Diagrammen

Alle Eingaben, sowohl Diagramme als auch Modelle, werden für den Ansatz in Graphen umgewandelt. Der erstellte Graph ist schleifenfrei, Kanten von einem Knoten zu sich selbst sind nicht erlaubt. Alle Kanten sind gerichtet und einem Kantentypen zugeordnet. Die Kantentypen sind *default* und *hierarchy*. Von einem Knoten zu einem anderen Knoten kann eine oder keine Kante eines Typs existieren. Zwei gegenläufige Kanten gleichen Typs sind erlaubt.

Für *default*-Kanten ist keine feste Semantik definiert. *default*-Kanten können verschiedene Beziehungen darstellen, zum Beispiel Abhängigkeiten oder Vererbung. Die allgemeine Annahme ist, dass die Kante von dem Nutzer zum Genutzten zeigt. Die Semantik einer *hierarchy*-Kante ist, dass der Ursprungsknoten in einer Hierarchie unter beziehungsweise im Zielknoten eingeordnet ist.

Graphen, welche aus Diagrammen erstellt wurden, werden als Diagramm-Graphen bezeichnet. Graphen, welche aus Modellen erstellt wurden, werden als Modell-Graphen bezeichnet.

Diagramm Der Ansatz verwendet die Boxen-und-Linien-Form von Diagrammen. Es wird also eine Menge von Boxen mit den darin beinhalteten Boxen und den ausgehenden Linien übergeben. Alle Boxen sind Knoten des Graphen. Alle Linien sind gerichtete *default*-Kanten zwischen zwei Knoten. Jeder Knoten erhält den Text der Box. Alle Boxen in einer anderen Box werden mit einer *hierarchy*-Kante mit der Box verbunden, welche sie enthält. In Abbildung 2.8 ist gezeigt, wie ein Diagramm als Graph dargestellt werden kann.

Der Graph beinhaltet alle Boxen und Kanten aus dem Diagramm in Abbildung 2.7 und die zusätzlichen *hierarchy*-Kanten. Das Diagramm besteht aus drei Elemente C1, C2, C3, welche miteinander verbunden sind und in P1 enthalten sind, welches wiederum mit P2 verbunden ist. Im Graphen sind alle Verbindungen enthalten, aber statt der Beinhaltung in P1 sind C1, C2, C2 über eine *hierarchy*-Kante mit P1 verbunden.

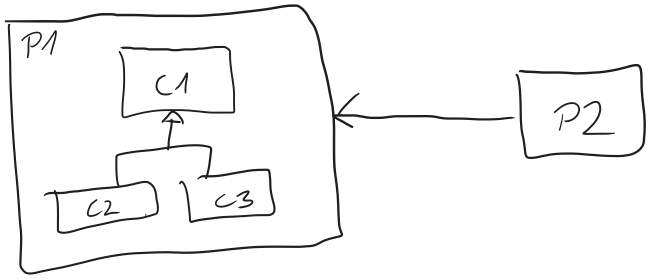

Abbildung 2.7: Ein Beispiel-Diagramm

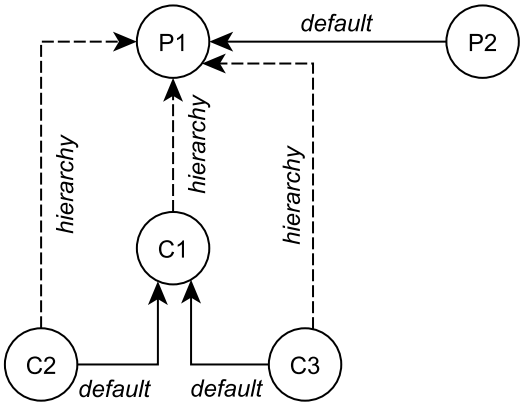

Abbildung 2.8: Die Abbildung des Beispiel-Diagramms auf einen Graphen

Die Transformation von Diagramm zu Graph ist invertierbar. Sie beruht auf zwei Operationen, die beide keinen Informationsverlust verursachen. Die Operationen sind in der Abbildung 2.9 dargestellt.

Architekturmodell Das Architekturmodell, siehe Unterabschnitt 2.4.1, enthält zwei für den Ansatz relevante Klassen. Diese sind Komponenten (*ArchitectureComponent*) und Schnittstellen (*ArchitectureInterface*). Alle Komponenten werden direkt auf Knoten im Graphen abgebildet. Im Architekturmodell hat jede Komponente eine Menge benötigter Schnittstellen und eine Menge bereitgestellter Schnittstellen. Für jede benötigte Schnittstelle einer Komponente wird eine *default*-Kante zu jeder Komponente erstellt, welche die Schnittstelle bereitstellt. Sollte eine Komponente eine Schnittstelle bereitstellen, die sie auch selbst benötigt, wird keine Kante erstellt, um Schleifen zu verhindern. Komponenten

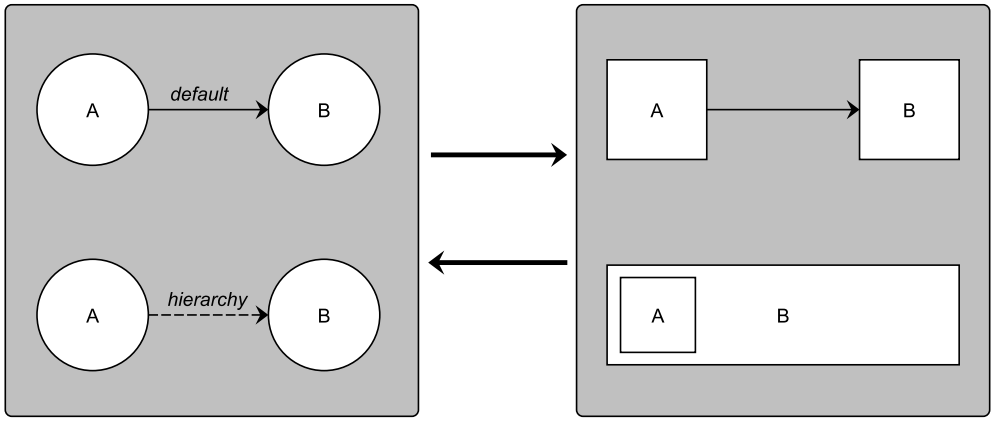

Abbildung 2.9: Transformation zwischen Diagramm und Graph

können Unterkomponenten haben. Dies wird dargestellt, indem die Unterkomponente eine *hierarchy*-Kante zur Oberkomponente hat.

Codemodell Die relevanten Klassen im Codemodell, siehe Unterabschnitt 2.4.2, sind Pakete (*CodePackage*) und Klassen sowie Schnittstellen, die beide Datentypen (*Datatype*) sind. Sowohl Pakete als auch Datentypen werden auf Knoten abgebildet. Datentypen können andere Datentypen erweitern oder implementieren. Diese Beziehungen werden als *default*-Kanten dargestellt, der erweiternde oder implementierende Datentyp ist der Ursprung der Kante. Alle Knoten von Paketen erhalten eine *hierarchy*-Kante zu dem Knoten des Paketes, welches sie enthält. Genauso werden für Datentypen *hierarchy*-Kanten zu den Paketen erstellt, welche sie enthalten. Sollte ein Datentyp innerhalb eines anderen Datentyps definiert sein, wird die *hierarchy*-Kante statt zu einem Paket zu dem enthaltenden Datentyp erstellt. Jeder Datentyp ist in einer Übersetzungseinheit (*CodeCompilationUnit*) enthalten, welche wiederum in einem Paket enthalten ist. Die Übersetzungseinheiten werden aber nicht im Diagramm dargestellt. Ein Grund dafür ist, dass insbesondere in Sprachen wie Java für jede Klasse eine Datei existiert und in der Regel auch nur eine Klasse in einer Datei definiert wird.

Wird das erweiterte Codemodell verwendet, so werden alle Referenzen von einem Datentypen auf einen anderen als *default*-Kanten dargestellt. Der nutzende Typ dabei ist der Ursprung der Kante, der referenzierte Typ das Ende der Kante. Es werden keine Schleifen erstellt, daher werden Referenzen eines Typen auf sich selbst ignoriert. Zusätzlich werden *default*-Kanten für alle Paket-Abhängigkeiten erstellt. Ein Paket A ist von einem anderen Paket B abhängig, wenn es mindestens einen Typen in A gibt, der von einem Typen in B abhängt. Daher ist es möglich, dass zwei Pakete gegenseitig voneinander abhängig sind.

3 Verwandte Arbeiten

In diesem Abschnitt werden verwandte Arbeiten zu den einzelnen Aspekten dieser Arbeit präsentiert. Zuerst werden verschieden Arbeiten aus dem Bereich des automatisierten Findens von Nachverfolgbarkeitsverbindungen betrachtet, da auch diese Arbeit in diesen Bereich fällt. Dann werden Arbeiten zum Thema *graph matching* betrachtet, da es in dieser Arbeit zum Finden Nachverfolgbarkeitsverbindungen eingesetzt wird. Zuletzt wird betrachtet, in welchen Arbeiten bereits *graph matching* zum Finden von Nachverfolgbarkeitsverbindungen genutzt wurde.

3.1 Nachverfolgbarkeitsverbindungen

Eine Vielzahl von Arbeiten befasst sich mit dem Finden von Nachverfolgbarkeitsverbindungen zwischen Software-Artefakten. Im Englischen wird dieses Ziel als *trace link recovery* bezeichnet [10].

SWATTR von Keim u. a. [17] stellt Verbindungen zwischen natürlichsprachlicher Architekturdokumentation und Architekturmodell her. SWATTR steht für 'SoftWare Architecture Text Trace link Recovery'. Dazu ist SWATTR als Pipeline aus mehreren Schritten aufgebaut, wobei Schritte selbst erweiterbar sind. Der erste Schritt ist die Extraktion von Text aus der Architekturdokumentation und aller relevanten Elemente aus dem Architekturmodell. Der zweite Schritt identifiziert Elemente im Text und nutzt dazu die im Architektur-Metamodell definierten Typen. Im dritten und letzten Schritt werden dann ähnliche Elemente aus dem Text und dem Architekturmodell miteinander verbunden. Eine Schwierigkeit beim Verbinden der Elemente ist dabei, dass die Bezeichnung in Text von der im Modell für dasselbe Element unterschiedlich sein kann. Im Gegensatz zu dieser Bachelorarbeit befasst sich SWATTR weder mit Quelltext noch mit Diagrammen. Gleichzeitig werden in beiden Arbeiten informelle und formelle Artefakte miteinander verbunden. Des Weiteren ist die SWATTR-Arbeit aufgrund des informellen Aspektes verwandt zu der hier präsentierten Arbeit. Diagramme besitzen auch einen natürlichsprachlichen und informellen Aspekt, insbesondere wenn diese nicht einer Spezifikation folgen.

Der Quelltext wird von Telge [27] berücksichtigt. In seiner Arbeit werden Verbindungen zwischen Architekturmodell und Quelltext hergestellt. Im ersten Schritt wird dazu ein Codemodell aus dem Quelltext extrahiert. Die bereits in verschiedenen Formaten vorhandenen Architekturmodelle werden in ein gemeinsames und reduziertes Format überführt, welches nur benötigte Elemente enthält. Dann werden im zweiten Schritt Heuristiken angewendet, um die Ähnlichkeit der Elemente in den Modellen zu bestimmen. Dabei

weisen die Heuristiken jedem Paar von Code- und Architektur-Element eine Konfidenz zu, die angibt, wie sicher eine Verbindung beider Elemente korrekt ist. Ein Teil der Heuristiken nutzt die Bezeichner der Elemente, um diese miteinander zu vergleichen. Auch die Struktur wird genutzt, zum Beispiel gibt es eine Heuristik, welche Unterklassen der gleichen Komponente wie ihrer Oberklasse zuordnet. Dabei werden aber keine Verbindungen zwischen zwei vorher unverbundenen Elemente erstellt. Die in dieser Bachelorarbeit verwendete Code- und Architektur-Metamodelle entstammen seiner Arbeit.

Kleffmann u. a. [18] befassen sich mit dem Verbinden von Elementen in Skizzen. In ihrer Arbeit stellen sie ein System aus mehreren interaktiven Whiteboards vor. Auf diesen werden kollaborativ verschiedene Skizzen erstellt. Zwischen den Elementen und Skizzen werden von einer Hintergrund-Anwendung Verbindungen gesucht. Dazu wird hauptsächlich der Vergleich von erkanntem Text und die Information, welche Dokumente gleichzeitig angezeigt wurden genutzt. Mit diesen Verbindungen kann dann die Konsistenz von Element-Annotationen überprüft werden. Zum Beispiel kann ein Element in einer Skizze als "unveränderbar" annotiert werden, dann wäre die Annotation "änderung nötig" für das gleiche Element in einer anderen Skizze nicht konsistent. An jedem Element, das Teil einer Verbindung ist, wird ein Symbol angezeigt, welches auch als Schaltfläche dient, um die Skizzen mit den verbundenen Elementen zu öffnen. Im Vergleich zu dieser Bachelorarbeit muss aber für ein Diagramm angegeben werden, welchen Typ es hat.

Jongeling u. a. [16] stellen einen Ansatz vor, welcher es ermöglicht, digital erstellte Architekturdiagramme, als Modell zu nutzen. Zusätzlich zum Diagramm wird in dem Ansatz eine zweite Zeichnung als Legende genutzt, die für grafische Elemente eine Bedeutung angibt. Ein Beispiel hierfür wäre, dass gelbe Rechtecke eine Komponente, oder gestrichelte Linien eine Abhängigkeit sind. Mit dieser Legende kann die in ein Architekturmodell mit Textform umgewandelt werden. Informationen aus der Zeichnung über nicht zugeordnete Elemente werden dabei erhalten, daher ist eine Konvertierung von Text zu Zeichnung auch möglich. Somit kann das Diagramm als Modell genutzt werden, sowohl Veränderungen in der Textform als auch am Diagramm können gleichermaßen genutzt werden. Der Ansatz ist aber nur für Diagramme geeignet, welche einheitlich genug beziehungsweise formell sind, dass eine Legende genutzt werden kann. Zusätzlich wird auch die Konsistenzüberprüfung zwischen Diagramm und Implementierung untersucht. Dazu wird die Struktur aber nicht berücksichtigt, sondern nur die Menge der Komponenten verglichen.

Panichella u. a. [23] verbinden Code-Elemente mit Artefakten wie zum Beispiel Anwendungsfallbeschreibungen. Dazu werden erst mit auf Text basierenden Verfahren anfängliche Verbindungen gesucht. Unter Zuhilfenahme von struktureller Information wie zum Beispiel Vererbungsbeziehungen werden dann weitere Verbindungen gesucht. Die Arbeit stellt fest, dass die Qualität der über die Struktur gefundenen Verbindungen stark von der Qualität der davor identifizierten Verbindungen abhängt. Daher werden die initial gefundenen Verbindungen erst von einem Softwareingenieur geprüft. Von dieser Bachelorarbeit unterscheidet sich die Arbeit unter anderem dadurch, dass über die Struktur keine Verbindungen zwischen zwei vorher unverbundenen Elementen aufgebaut werden. Stattdessen werden immer nur weitere Code-Elemente zum gleichen Artefakt verbunden.

Zusätzlich ist in dieser Bachelorarbeit kein manueller Zwischenschritt zur Überprüfung vorgesehen.

3.2 Graph Matching

Das Ziel von *graph matching* ist es, für die Knoten eines Graphen übereinstimmende Knoten in einem anderen Graphen zu finden. Übereinstimmend bedeutet hier, dass die Nachbarschaft bezüglich Knoten und Kanten strukturell möglichst ähnlich ist. Es wird unterschieden zwischen Algorithmen für exakte und approximative Übereinstimmung. Bei exakten Algorithmen wird auch von Graph-Isomorphismen gesprochen, dieses Gebiet ist eher von theoretischer Bedeutung [30].

3.2.1 Allgemeine Algorithmen

Für diese Bachelorarbeit werden approximative Algorithmen benötigt. Dies ist schon allein deshalb der Fall, da durch Inkonsistenzen eine exakte Übereinstimmung nicht möglich ist. Des Weiteren ist die Art der behandelten Graphen zu beachten. Im Bereich des maschinellen Sehens wird auch *graph matching* angewendet [19]. Hier operieren die Algorithmen teils auf reiner Struktur und berücksichtigen nicht Kanten- und Knotenattribute. Diese Information ist aber im Bereich der Softwaretechnik von Bedeutung, zum Beispiel in der Form von Elementbezeichnern.

Zusätzlich ist die genaue Aufgabenstellung der Algorithmen zu beachten. Im Bereich des maschinellen Sehens und sozialen Netzen sind die Algorithmen etwas spezialisierter. Statt nach Übereinstimmungen zwischen zwei Graphen zu suchen, wird in diesen Bereichen eine Mustersuche durchgeführt. In einem großen Datengraph wird also nach Übereinstimmungen zu einem kleineren Anfrage-Graphen gesucht. Ein Beispiel für einen solchen Algorithmus ist MAGE von Pienta u. a. [24]. MAGE ist entworfen, um in einem Graphen mit Knoten- und Kantenattributen mit mehreren Millionen Knoten und mehreren hundert Millionen Kanten Subgraphen zu finden. Dabei kann der Subgraph Platzhalter beinhalten, sodass an deren Stelle keine bestimmten Attribute erwartet werden. Da Diagramme teils einen Ausschnitt des Gesamtprojektes darstellen, könnten die Aufgabenstellung in diesen Fällen als Anfrage dargestellt werden. Doch Diagramme können auch gesamte Modelle abbilden und die Semantik einer Anfrage ist dann nicht klar gegeben. Auch ist nicht direkt klar, welche Knoten der Anfrage als Platzhalter markiert werden sollten. Zusätzlich sind diese Algorithmen für viel größere Graphen ausgelegt, als es für Softwaremodelle zu erwarten ist.

Im Vergleich dazu gibt es im Bereich des Modell- und Relations-Abgleiches Algorithmen, welche tendenziell mit zwei etwa gleich großen Graphen arbeiten. Ein solcher Algorithmus ist der *similarity flooding*-Algorithmus [21], welcher der Datenbankforschung enstammt. Der Algorithmus ist für den Abgleich von Modellen, unabhängig vom Anwendungsbereich,

konzipiert. Zwar ist die Ausgangssituation zweier etwa gleich großer Graphen im Anwendungsfall dieser Bachelorarbeit nicht garantiert, dies ist aber keine formale Voraussetzung des Algorithmus. Der Algorithmus wurde bereits für die Suche nach Nachverfolgbarkeitsverbindungen genutzt, darauf wird im folgenden Abschnitt eingegangen.

3.2.2 Algorithmen zum Finden von Nachverfolgbarkeitsverbindungen

Es gibt bereits Arbeiten, welche *graph matching* zum Finden von Nachverfolgbarkeitsverbindungen und für modellgetriebene Entwicklung nutzen.

Grammel, Kastenholz und Voigt [11] untersuchen das Finden von Nachverfolgbarkeitsverbindungen zwischen Start- und Zielmodell einer Modelltransformation. Dabei verwendete Werkzeuge zur Modelltransformation können teilweise bereits Nachverfolgbarkeitsverbindungen während oder nach der Transformation erstellen. Doch nicht immer können *trace links* automatisch erstellt werden, da zum Beispiel die betrachteten Modelle auf einem unterschiedlichen Abstraktionsniveau angeordnet sind. Daher wird ein Ansatz vorgestellt, der beliebige Modelle unabhängig von den darin verwendeten Sprachen verbinden kann. Der Ansatz ist in vier Schritte aufgeteilt. Als Erstes werden die beiden Modelle geladen und Modelle sowie Metamodelle in ein gemeinsames Datenmodell-Format konvertiert, das eine Graph-Struktur aufweist. Im zweiten Schritt werden die beiden Modelle verglichen, das Ergebnis davon ist eine Matrix mit Ähnlichkeits-Werten für alle Paare von Elementen in den beiden Modellen. Verschiedene Algorithmen können hier kombiniert werden. Diese Algorithmen nutzen die Struktur der Graphen sowie Techniken des *schema matching* aus, um die Ähnlichkeit zu bestimmen. Dann werden im dritten Schritt die Ähnlichkeits-Matrizen aller ausgeführten Algorithmen kombiniert, um die endgültigen Werte zu bestimmen. Zuletzt werden dann mit diesen Werten die *trace links* bestimmt.

Heraguemi, Abid und Amirat [13] untersuchen das Verbinden von verschiedenen Architekturmodellen. Es werden Architekturmodelle betrachtet, welche die gleiche Architektur in verschiedenen Metamodellen darstellt. Das Ziel von ihnen ist es, die Transformation eines Modells zwischen verschiedenen Metamodellen zu vereinfachen. Statt Transformationen manuell zu erstellen, soll durch das Verbinden von Modellen automatisch eine Transformation gefunden werden. Die Verbindungen stellen dann Abbildungen von einem Modell in das andere dar und liefern so Transformationsregeln. Das eigentliche Verbinden der Modelle wird mit dem Algorithmus *similarity flooding* durchgeführt. Um den Algorithmus anwenden zu können, müssen beide Modelle zu einem gerichteten Graphen mit Kantenattributen transformiert werden. In der Arbeit wird konkret das Verbinden zweier Architekturmodelle betrachtet, doch um das Ziel zu erreichen, könnten auch die Metamodelle selbst verbunden werden.

Beide Arbeiten wenden die Graph-Algorithmen auf Paare von Modellen an. Das Verbinden von Diagramm und Modell wird von den gennanten Arbeiten also nicht behandelt.

4 Betrachtete Diagramme

Während der Implementierung des Ansatzes sind Eingabedaten nötig, mit denen der Ansatz überprüft und verbessert werden kann. Entscheidungen bei der Implementierung der Teilschritte können auf Basis der Qualität getroffen werden. Dabei wird die Qualität anhand des F1-Scores gemessen.

Für diesen Ansatz werden sowohl reale als auch synthetische Diagramme verwendet. Im Folgenden werden die verwendeten realen Diagramme und deren Ursprung dargestellt. Dann wird beschrieben, wie synthetische Diagramme erstellt werden.

4.1 Reale Diagramme

Zur Auswertung werden die Diagramme der Projekte im Benchmark-Repository von ArDoCo [5] genutzt. Diese Projekte beinhalten jeweils ein Architekturmodell und ein Codemodell. Die verwendeten Diagramme sind in Abbildung 4.1 aufgelistet. Bedingungen an die Projekte sind, dass diese Diagramme enthalten, welche Code- oder Architektur darstellen. Diese Diagramme sollten informell sein, und nicht zum Beispiel vom Quelltext generierte UML-Diagramme. Zusätzlich müssen die Projekte Quelltext und Architekturmodell verfügbar machen. Ein Grund, weshalb weitere real existierende Architektur-Diagramme nicht genutzt werden konnten ist, dass die Projekte keine Architekturmodelle enthielten. Für jedes Diagramm wird manuell die Boxen-und-Linien-Form sowie Goldstandards für die Schritte des Ansatzes erstellt.

`BIG_BLUE_BUTTON`	Architekturdiagramm [6], Kürzel: `BBB`, siehe Abbildung 1.1
`TEAMMATES_ARCHITECTURE`	Architekturdiagramm [26], Kürzel: `TM_A`
`TEAMMATES_PACKAGES`	Codediagramm [26], Kürzel: `TM_P`
`TEAMMATES_UI`	Codediagramm [26], Kürzel: `TM_U`
`TEA_STORE`	Architekturdiagramm [28], Kürzel: `TS`
`MEDIA_STORE`	Architekturdiagramm [25], Kürzel: `MS`

Abbildung 4.1: Die verfügbaren Benchmark-Diagramme

4.2 Synthetische Diagramme

Mit dem Benchmark-Repository sind nur eine geringe Anzahl an Projekten mit Diagrammen und Architekturmodell verfügbar. Eine Aufteilung der vorhandenen Diagramme in eine Implementierungs- und eine Evaluations-Menge ist daher nicht erreichbar. Es wäre möglich, als Eingabe während der Implementierung ausschließlich die realen Diagramme zu nutzen, welche auch zur Evaluation verwendet werden. Um eine Überanpassung an die vorhandenen realen Diagramme zu vermeiden und einen allgemeinen Ansatz zu erstellen, werden während der Implementierung zuerst synthetische Diagramme verwendet. Diese synthetischen Diagramme sind für das Gebiet der Inkonsistenz-Erkennung weniger relevant als informelle und von Hand erstellte Diagramme, denn generierte Diagramme brauchen nicht konsistent gehalten zu werden – sie können neu generiert werden. Generierte Diagramme folgen der Spezifikation des generierenden Algorithmus, sie sind also nicht informell. Zusätzlich beruhen synthetische Diagramme auf vereinfachten Annahmen. Für die finale Evaluation sowie Zwischenevaluationen des Ansatzes werden daher reale Diagramme verwendet. Ein Vorteil von synthetischen Diagrammen ist Kontrolle über die Anzahl von Inkonsistenzen in der Eingabe. Dies ist hilfreich, da in den für die Evaluation betrachteten realen Diagrammen Inkonsistenzen in verschiedener Häufigkeit und Art auftreten.

4.2.1 Generierung von synthetischen Diagrammen

Zur Darstellung der Generierung wird in den folgenden Abschnitten das Codemodell in 4.1 verwendet. Der vereinfachte Code beinhaltet drei Klassen, welche im Packet *io* enthalten sind. Dieser Code steht stellvertretend für ein Codemodell, welches die selben Informationen wie der Code enthält.

Listing 4.1: Vereinfachter Code, als Repräsentation des Codemodells

```
package io;

class Stream;
class FileStream extends Stream;
class MemoryStream extends Stream;
```

Für ein gegebenes Modell wird die Boxen-und-Linien-Form eines Diagramms erstellt. Dies ist analog zur Transformation eines Modells zu einem Graphen, dargestellt in Absatz 2.8 und Absatz 2.8. Dabei wird keine visuelle Repräsentation erstellt, sondern eine Datenstruktur, welche die Boxen und Linien enthält.

Anstelle von Knoten werden Boxen erstellt. Die Rolle von *default*-Kanten übernehmen Linien. Wird in der Transformation zu einem Graphen eine *hierarchy*-Kante erstellt, so wird dies im Diagramm durch Beinhaltung dargestellt. Für eine *hierarchy*-Kante von A zu B bedeutet dies, dass A in B enthalten ist.

Zusätzlich werden Verbindungen zwischen den Elementen in Diagramm und Modell erstellt. Eine Verbindung ist ein Tupel von Diagram-Element und Modell-Element, welche dasselbe repräsentieren. Alle Verbindungen werden gespeichert, dies sind die vorhandenen Verbindungen und dienen als Goldstandard für spätere Untersuchungen. In Abbildung 4.2 ist das Codemodell 4.1 als Graph und Diagramm dargestellt. Die generierten synthetischen Diagramme besitzen die gleiche Struktur wie dargestellt, werden aber nicht visuell, sondern als Datenstruktur verwendet.

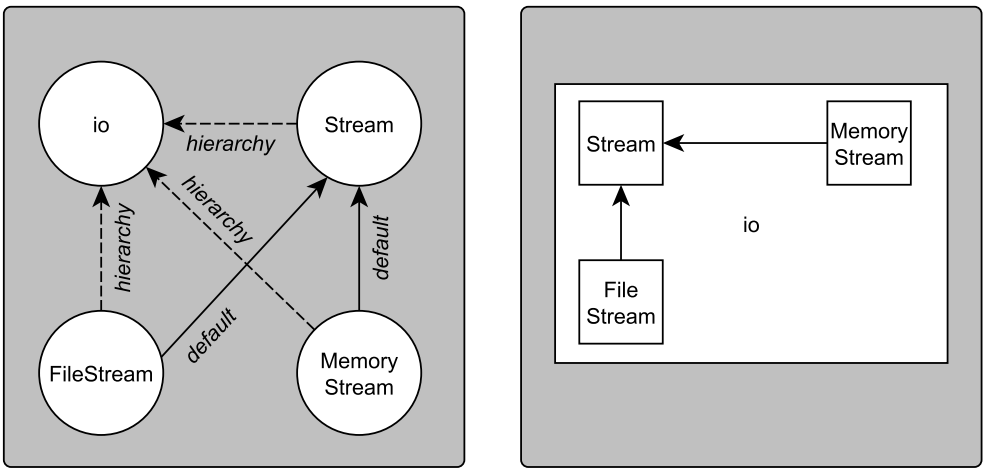

Abbildung 4.2: Das Codemodell als Graph und Diagramm

Die Struktur des Modell-Graphen und des daraus generierten Diagramms stimmt stark überein und beinhaltet die gleiche Information. Für die Verbindungssuche wird das Diagramm wieder in einen Graphen transformiert, der identisch zum Modell-Graphen ist, sofern das Diagramm nicht verändert wurde. Dies ist in Abbildung 2.9 dargestellt. Die Suche nach sich entsprechenden Elementen in Modell und synthetischem Graphen ist also einfach. Daher eignen sich unveränderte synthetische Diagramme nur begrenzt für weitere Untersuchungen. Durch die Veränderung der synthetischen Diagramme kann die Auswirkung dieser Änderungen auf die Qualität der Verbindungssuche untersucht werden. Zusätzlich können synthetische Änderungen auch auf reale Diagramme angewendet werden, um die Anzahl an Inkonsistenzen zu erhöhen.

4.2.2 Synthetische Änderungen

Ausgehend von den betrachteten Arten von Inkonsistenzen in Abschnitt 2.6 werden synthetische Änderungen definiert. Die synthetischen Änderungen sind auf Graphen definiert, da Graphen die gemeinsame Datenstruktur sind, in die Diagramme und Modelle transformiert werden. Der zu verändernde Graph kann aus einem Diagramm oder Modell generiert werden. Für ein Tupel aus zusammengehörendes Diagramm und Modell werden die Änderungen nur auf einer der Seiten angewendet, um Inkonsistenzen zu erzeugen.

Einige der Änderungen verändern zusätzlich zum Graphen auch die vorhandenen Verbindungen zwischen Diagramm und Modell. Wie bereits in Absatz 2.8 beschrieben, ist die Transformation von Diagramm zu Graph invertierbar. Durch Transformation eines Diagramms zu einem Graphen, anschließender Anwendung der Änderungen und zuletzt invertierter Transformation zu einem Diagramm existiert eine Operation zur Anwendung von Änderungen auf Diagramme. Es ist nicht möglich auf die gleiche Weise Änderungen auf ein Modell anzuwenden, da keine Transformation von Graph zu Modell existiert. Der Grund hierfür ist, dass bei der Transformation von Modell zu Graph Informationen verloren gehen.

Die Änderungsklassen aus Abschnitt 2.6 haben die folgenden Auswirkungen auf den Graphen:

- *move*: Ein Knoten wird innerhalb der Hierarchie verschoben. Alle *hierarchy*-Kanten, die von diesem Knoten ausgehen, werden entfernt. Es wird ein neuer Knoten gewählt, zu dem eine *hierarchy*-Kante erstellt wird. Dies hat keine Auswirkungen auf die Verbindungen zwischen Graph und Modell.
- *rename*: Ein Knoten wird umbenannt. Dies hat keine Auswirkungen auf die Verbindungen zwischen Graph und Modell.
- *create*: Ein neuer Knoten wird erstellt. Der Knoten kann eine *hierarchy*-Kante zu einem anderen Knoten erhalten. Zusätzlich werden eine geringe Anzahl von *default*-Kanten von dem neuen Knoten zu anderen Knoten sowie in umgekehrter Richtung erstellt. Da Beziehung als eine Eigenschaft der Knoten betrachtet werden, werden nur neue Kanten zu dem neuen Knoten als Inkonsistenz betrachtet. Der neue Knoten hat keine Verbindung zum Modell.
- *delete*: Ein Knoten wird gelöscht. Zusätzlich werden alle Kanten von und zu dem Knoten gelöscht. Die Verbindung des Knotens zum Modell wird gelöscht.
- *connect*: Zwischen zwei Knoten wird eine *default*-Kante erstellt.
- *disconnect*: Eine *default*-Kanten zwischen zwei Knoten wird gelöscht.

In Abbildung 4.3 sind zwei beispielhafte Änderungen dargestellt. Die Änderungen *rename* und *delete* sind auf das Diagramm angewendet, die veränderten Elemente sind in Rot dargestellt. Zusätzlich zur Änderung am Diagramm muss in diesem Beispiel auch die Verbindung von `MemoryStream` zum entsprechenden Element im Modell gelöscht werden. Somit besteht nicht mehr die Erwartung, dass eine Verbindung gefunden wird, welche `MemoryStream` beinhaltet. Das Fehlen von `MemoryStream` ist dennoch eine Inkonsistenz, da ein entsprechendes Element im Modell existiert und die Änderung unvollständig angewendet wurde.

Wie bereits erläutert, ist es mit der dargestellten Methodik nicht möglich, die synthetischen Änderungen direkt auf ein Modell anzuwenden. Die Anwendung von *refactorings* auf Diagramme, während das Modell unverändert bleibt, ist so nicht realistisch. Das zeigt sich darin, dass *refactorings* als Änderungen von Code und Architektur definiert sind [7]. In realen Anwendungsfällen werden *refactorings* primär auf die Modelle beziehungsweise den

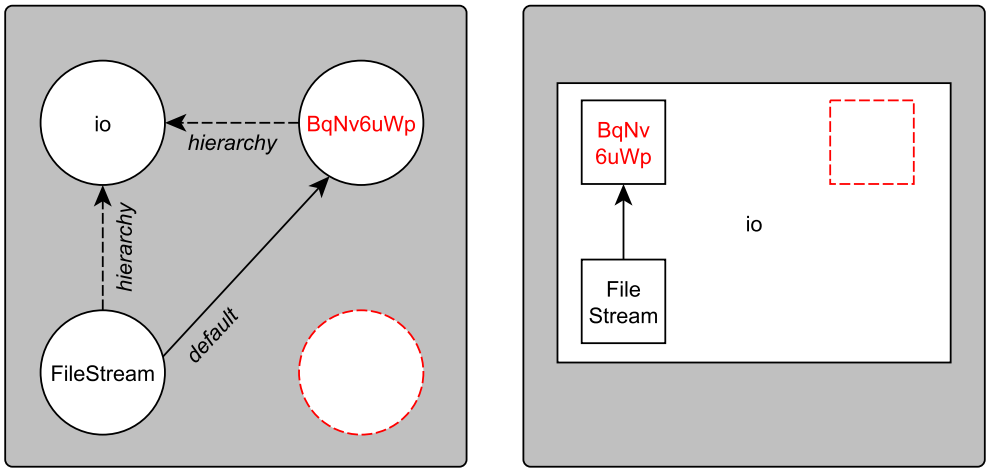

Abbildung 4.3: Synthetische Änderungen angewendet auf Graph und Diagramm, Rot: Auswirkung der Änderungen

Code und nicht auf die Diagramme angewendet. Eine Analyse von Baltes und Diehl [4] beschreibt, dass einige Softwareentwickler ihre Diagramm-Skizzen nicht als primäres Artefakt betrachten. Sobald das Diagramm in Code überführt wurde, wird es als ersetzt angesehen. Zusätzlich wird der Code weiter entwickelt, sodass das Diagramm nicht mehr aktuell ist.

Dennoch ist es für die Untersuchung während der Implementierung ausreichend, die *refactorings* auf Diagramme anzuwenden. Insbesondere für die Suche nach Verbindungen ist es nicht relevant, ob die Änderungen auf das Diagramm oder das Modell angewendet werden. Die Definition des Algorithmus Similarity Flooding behandelt beide an diesen übergebenen Graphen gleich. Ausgehend von der formalen Definition, ist das Ergebnis bei Tausch der Eingaben gleich. Bei Ausführung von Similarity Flooding ist wegen Implementierungseigenschaften das Ergebnis je nach geänderter Seite unterschiedlich. Die Unterschiede der beobachten Ergebnisse sind jedoch gering und innerhalb der durch synthetische Änderungen bedingten Schwankungen. Für alle folgenden Untersuchungen werden daher alle Änderungen auf die Diagramm-Seite angewendet. Dies ermöglicht es, synthetische Diagramme auf die gleiche Art wie reale Diagramme als Eingabe an die Pipeline zu übergeben. Für Modelle wäre dies so nicht möglich, da die definierten Änderungen nicht auf Modelle anwendbar sind.

4.2.3 Betrachtung der zugrundeliegenden Vereinfachungen

Die im Vorherigen erläuterte Erstellung von synthetisierten Diagrammen basiert auf Vereinfachungen, die für reale Diagramme nicht immer zutreffend sind. In diesem Abschnitt werden zwei dieser Vereinfachungen betrachtet und deren Auswirkungen untersucht.

Die erste Vereinfachung ist, dass ein Diagramm das gesamte Modell darstellt. Für die großen Codemodelle ist dies nicht unbedingt der Fall. Es wird *graph matching* zur Verbindungssuche auf Diagramme angewendet, welche nur Ausschnitte des Modells darstellen. Zur Erstellung eines solchen Diagramms wird erst ein komplettes Diagramm synthetisiert. Dann wird ein Blatt-Knoten gewählt und die Hierarchie nach oben traversiert, bis die Hierarchie unter dem jetzigen Knoten mehr als 5% und weniger als 25% des gesamten Modells enthält. Auf dieses Diagramm werden dann synthetische Änderungen angewendet. Dabei zeigt sich, dass die Suche nach Verbindungen durch das Betrachten von Ausschnitten erschwert wird. Die Qualität der gefundenen Verbindungen, gemessen am F1-Score, sinkt. Für die Änderungsklassen *move* und *create* ist der Rückgang am größten. Gleichzeitig ist zu erkennen, dass die Wahl der Parameter eine Auswirkung auf die Qualität hat. Es ist nicht auszuschließen, dass es eine Wahl der Parameter gibt, welche besonders günstig für Ausschnitte ist.

Die zweite Vereinfachung ist, dass die Namen der Diagrammelemente exakt den Namen der Elemente im Modell entsprechen. Es wird *graph matching* zur Verbindungssuche auf Diagramme angewendet, in denen alle Namen ein Suffix haben. Dazu wird ein komplettes Diagramm synthetisiert. Dann wird an den Namen jeder Box der Suffix `Impl` angehängt. Auf dieses Diagramm werden dann die synthetischen Änderungen angewendet. Die Qualität der Verbindungssuche ist dann um bis zu 0,1 geringer, in den meisten Fällen aber weniger. Das Anhängen eines Suffixes bedeutet für Textähnlichkeit mit Levenshtein-Distanz, dass alle Begriffe eine Mindestähnlichkeit erhalten. Würde stattdessen ein zufälliger Text angehängt, wäre dies vergleichbar mit einer Maximalähnlichkeit aller Begriffe. Durch Anpassung von Schwellenwerten kann dieser Änderungen der Bezeichner also gegengesteuert werden.

Werden sowohl Ausschnitte betrachtet und Suffixe verwendet, so wird die Qualität der Verbindungssuche durch die Kombination nicht weiter verschlechtert. Mit der realistischeren Synthetisierung sind keine grundlegenden Unterschiede zu erkennen, die eine Anpassung des Ansatzes benötigen. Es bestehen weitere Vereinfachungen, die möglicherweise grundlegende Unterschiede zwischen synthetischen und realen Diagrammen verursachen. Weitere Schritte zu einer realistischeren Synthetisierung werden hier aber nicht weiter betrachtet.

5 Aufbau und Implementierung der Pipeline des Ansatzes

Um das Ziel zu erreichen, den Diagrammtyp zu ermitteln und dann Inkonsistenzen zu finden, wird der im Folgenden beschriebene Ansatz genutzt.

Der Ansatz ist in drei Schritte unterteilt. Der erste Schritt ermittelt für jeden bekannten Diagrammtypen, wie sehr das behandelte Diagramm diesem entspricht. Im zweiten Schritt werden Verbindungen zwischen dem Diagramm und allen zu den Diagrammtypen passenden Modellen durchgeführt. Im dritten und letzten Schritt wird dann auf Basis der Verbindungen nach Inkonsistenzen gesucht.

Als Eingabe wird ein Diagramm aus einem Softwareprojekt auf der einen Seite und auf der anderen Seite Modelle des Softwareprojektes benötigt. Die genutzten Modellarten sind Codemodelle auf Basis des Quelltexts und Architekturmodelle. Abbildung 5.1 zeigt die benötigten Eingaben, also das Codemodell, das Architekturmodell und der Diagramm-Graph. Auch die bereits aufgezählten drei Schritte sind im Diagramm abgebildet. Diagramme (zum Beispiel als Bild) werden nicht direkt als Eingabe akzeptiert, sondern müssen in der Boxen-und-Linien-Form vorliegen. Wie zu erkennen ist, benötigen alle drei Schritte die Modelle und das Ergebnis des vorherigen Schrittes.

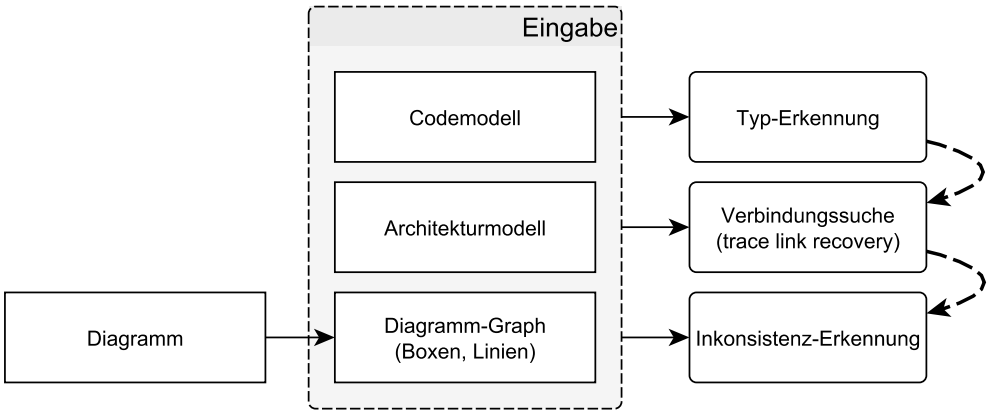

Abbildung 5.1: Überblick über die Schritte und dafür benötigte Eingaben

Die Implementierung nutzt die Pipeline-Architektur aus ArDoCo [3]. Die Schritte des Ansatzes werden direkt als Schritte der Pipeline implementiert. Pipeline-Schritte sind Agenten, welche aus einer Menge von Informanten bestehen.

Vor der Ausführung der drei primären Schritte des Ansatzes werden Schritte zur Vorbereitung ausgeführt. Die Vorbereitung beinhaltet das Laden des Diagramms und der Modelle. Dazu wird ein in ArDoCo [3] vorhandener Agent zum Laden beider Modelle verwendet. Ein weiterer, neu erstellter, Agent lädt das Diagramm.

5.1 Typ-Erkennung

Die Aufgabe der Typ-Erkennung ist es, zu entscheiden, welche der Modelle mit dem Diagramm an die Verbindungssuche übergeben wird. Dazu muss ermittelt werden, welches Modell vom Diagramm repräsentiert wird. Die Typ-Erkennung kann für ein Diagramm auch mehrere Modelle auswählen.

Diese Arbeit beschränkt sich auf Diagrammtypen, welche die Struktur von Code oder Architektur beschreiben. So sind Diagramme wie Sequenzdiagramme durch ihre zeitliche Dimension nicht direkt auf eine Boxen-und-Linien-Form abbildbar. Viel wichtiger ist hier aber, dass die genutzten Modelle die benötigte Information über zeitliche Abläufe nicht zur Verfügung stellen. Die behandelten Diagrammtypen in dieser Arbeit sind die folgenden:

- Architekturdiagramm: Zeigt Komponenten und deren Verwendungsbeziehungen.
- Codediagramm: Zeigt Klassen und Pakete, sowie deren verschiedenen Beziehungen.

Der Diagrammtyp stellt keine Formalisierung des Diagrams dar. Es ist lediglich eine Abschätzung darüber, welches Modell im Diagramm dargestellt wird, also über den Zweck der Abbildung. Die Typisierung beruht auf der Bedingung, dass Diagramme nur maximal eines der Modelle darstellen. Zwar können mehrere Typen ausgewählt werden, in den weiteren Schritten werden die gewählten Typen aber weiter einzeln betrachtet.

5.1.1 Aufbau der Typ-Erkennung

Der Schritt der Typ-Erkennung besteht aus drei Teilen. Im ersten Schritt werden alle Wörter in den geladenen Modellen gezählt. Im zweiten Schritt werden für alle Boxen mögliche Vorkommen in den Modellen gesucht, im dritten dann die Übereinstimmung des Diagramms mit den Modellen berechnet. Die einzelnen Schritte werden im Folgenden weiter beschrieben.

Worthäufigkeit Der erste Teil berechnet für jedes Wort in jedem Modell, wie häufig es in dem Modell vorkommt. Dazu werden alle Bezeichner im Modell extrahiert und in Worte aufgeteilt. Wortgrenzen werden anhand von nicht-alphabetischen Zeichen und dem Wechseln von Klein- zu Großschreibung erkannt. Dieser Schritt dient dazu, für verwendete Textähnlichkeitsfunktionen notwendige Daten zur Verfügung zu stellen, die zur Gewichtung nach Worthäufigkeit nötig sind. Nicht alle Textähnlichkeitsfunktionen der Typ-Erkennung benötigen diese Daten, da aber auch die Verbindungssuche diese Daten nutzt, ist dieser Teil nicht optional.

Suche nach Vorkommen Im zweiten Teil wird für jede Box im Diagramm nach Hinweisen gesucht, dass diese in einem Modell vorkommt. Dazu werden bereits einfache Nachverfolgbarkeitsverbindungen zwischen dem Diagramm und den Modellen gesucht. Eine Box kommt in einem Modell vor, wenn es im Modell ein Element gibt, welches einen ähnlichen Namen hat. Die Ähnlichkeit wird anhand einer beliebigen Textähnlichkeitsfunktion berechnet, die berechnete Ähnlichkeit muss über einem Schwellenwert liegen. Verschiedene Textähnlichkeitsfunktionen können über einen Parameter gewählt werden. Für alle folgende Untersuchungen wird, sofern nicht anders beschrieben, die Levenshtein-Ähnlichkeit verwendet. Für jede Box werden alle gefundenen Vorkommen gespeichert, auch mehrere Vorkommen in einem Modell sind möglich. Zusätzlich wird für jedes Vorkommen die durch die Box abgebildete Rolle gespeichert. Im Architekturmodell gibt es die Rollen *architecture:component* und *architecture:interface*. Im Codemodell gibt es die Rollen *code:package*, *code:class* und *code:interface*.

Spezifikationen wie UML [1] definieren konkretere Diagrammtypen als hier verwendet. So existieren statt Codediagrammen die Typen Klassendiagramm und Paketdiagramm. Eine genauere Erkennung solcher Typen wäre mithilfe der zugeordneten Rollen denkbar. Der Ansatz führt eine solche genaue Typisierung aber nicht durch, denn nicht alle realen Diagramme sind klar einem Untertypen zugeordnet. So ist es möglich, dass in einem Diagramm weitere Diagramme anderen Types eingebettet sind. Ein Beispiel hierfür ist ein Paketdiagramm, in dem die Pakete Klassendiagramme beinhalten. Durch die Verwendung der abstrakten, modellbezogenen Diagrammtypen statt spezifischer Typen wird eine große Anzahl an zu behandelnden Mischfällen umgangen. Dies ist nötig, da die behandelten Diagramme informell sind und keiner Spezifikation folgen.

Der verwendete Schwellenwert der Textähnlichkeit ist abhängig vom Modell-Typen. Da Code- und Architekturmodell unterschiedliche Eigenschaften aufweisen, liegt es nahe, beide Fälle unterschiedlich zu behandeln. Zu diesen unterschiedlichen Eigenschaften gehören anderen Namenskonventionen, unterschiedliche Größe und andere Struktur. Statt einem gemeinsamen Schwellenwert-Parameter wird daher der Parameter `similarity_threshold_architecture` für Architekturdiagramme und der Parameter `similarity_threshold_code` für Codediagramme eingeführt.

Die Typ-Erkennung betrachtet nur Elemente und keine Beziehungen im Modell. Daher ist es nicht relevant, ob das originale Codemodell (siehe Unterabschnitt 2.4.2) oder erweiterte Codemodell (siehe Abschnitt 2.5) genutzt wird.

Berechnung der Übereinstimmung Der dritte Teil berechnet für beide Modelle den Anteil der Diagramm-Elemente, die darin mindestens ein Vorkommen haben. Dazu werden die im ersten Teil gespeicherten Vorkommen verwendet. Der Anteil wird hier als Übereinstimmung bezeichnet. Das Modell mit der höheren Übereinstimmung wird als Typ des Diagramms ausgewählt. Sollte die Übereinstimmung für beide Modelle geringer als der Schwellenwert `match_threshold` sein, wird sich für keines der Modelle entschieden. Liegen die Übereinstimmungen beider Modelle nahe beieinander, das heißt der Unterschied ist

kleiner als der Schwellenwert `match_difference`, werden beide Modelle gewählt. Die folgenden Schritte werden für jedes gewählte Modell ausgeführt und alle Ergebnisse separat geführt.

Abbildung 5.2 zeigt ein einfaches Beispiel für berechnete Übereinstimmungen. Das Diagramm enthält vier Boxen *A*, *B*, *C* und *D*. Jeder Buchstabe steht stellvertretend für eine Gruppe von Bezeichnern, die sich sehr ähnlich sind. Im Architekturmodell gibt es für drei der vier Boxen im Diagramm ein Vorkommen. Daher ist die Übereinstimmung 75%. Im Codemodell werden hier zwar insgeasmt vier Vorkommen gefunden, doch diese entsprechen nur zwei verschiedenen Boxen im Diagramm. Daher ist die Übereinstimmung 50%.

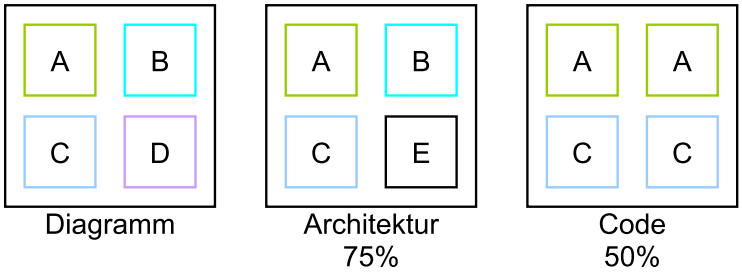

Abbildung 5.2: Einfaches Beispiel für berechnete Übereinstimmung

5.1.2 Anwendung auf synthetische Diagramme

Die synthetischen Diagramme sind für den Einsatz bei der Entwicklung der Verbindungssuche und Inkonsistenz-Erkennung gedacht. Dennoch ist eine Nutzung der synthetischen Diagramme als Eingabe bei der Typ-Erkennung prinzipiell möglich. Dabei ist zu beachten, dass es in dieser Anwendung einige Faktoren gibt, welche die Nutzbarkeit der synthetischen Diagramme einschränken. Für jedes synthetische Diagramm sind alle erwarteten Verbindungen zum ursprünglichen Modell bekannt. Es sind aber keine Informationen über potenzielle Übereinstimmungen mit anderen Elementen im gleichen Modell vorhanden. Genauso sind keine Informationen vorhanden, ob sinnvolle Vorkommen in anderen Modellen vorhanden sind. Zusätzlich sind die Namen der Elemente im Modell und daraus generierten Diagramm identisch, was eine Suche nach Vorkommen stark vereinfacht. Um dies zu beheben, werden an alle Namen der Boxen der Suffix `Impl` angehängt. In Code-Diagrammen wird nur ein Ausschnitt des Modells verwendet. In den Ergebnissen der folgenden Untersuchung zeigt sich, dass auch mit diesen Techniken die Namen im zugehörigen Modell viel ähnlicher als die im anderen Modell sind.

Als Textähnlichkeitsfunktion wurde hier die Levenshtein-Ähnlichkeit verwendet. Für den Ursprungs-Modelltypen ist die Übereinstimmungen sehr hoch (über 70%) und für den anderen Modelltypen geringer. Somit trifft der Ansatz die richtige Entscheidung, was mit den identischen beziehungsweise ähnlichen Namen begründet werden kann.

Synthetische Architekturdiagramme Abhängig vom Codemodell schwankt die Übereinstimmungen zwischen synthetischen Architekturdiagramm und Modell zwischen unter 25% und um 80%. Der Grund hierfür ist vermutlich, dass zwei der Codemodelle sehr große Projekte mit vielen Klassen beschreiben, während die anderen beiden kleinere Projekte beschreiben. Die größeren Projekte besitzen einzelne Pakete, die namentlich der Architektur entsprechen, der Großteil der Klassen darin ist aber entsprechend der speziellen Aufgabe benannt. Die kleineren Projekte folgen stärker der Architektur, Komponenten werden darin durch einzelne Klassen abgebildet. Daher stimmen deren Bezeichner stärker mit denen im Architekturdiagramm überein. Für Architekturdiagramme sind 50% der gefundenen Vorkommen unerwartet, was durch die Dopplung von Komponente und Schnittstelle verursacht wird. In den betrachteten Architekturmodellen gibt es für den Großteil der Komponenten eine Schnittstelle mit ähnlichem Namen.

Synthetische Codediagramme Die ermittelten Übereinstimmungen von synthetischen Codediagrammen und Architekturmodell sind mit Werten unter 50% gering. Für die großen Codemodelle kann dies wieder mit dem Aufbau erklärt werden. In den kleinen Codemodellen ist der Grund womöglich, dass nur Ausschnitte des Modells verwendet werden und daher nur ein Teil der namentlich ähnlichen Elemente im Diagramm vorhanden sind. Mehr als die Hälfte der Vorkommen sind unerwartet, ähnlich wie bei Architekturdiagrammen können gleich benannte Schnittstellen und Klassen der Grund sein. Ein weiterer Faktor, der dies verursacht, ist die Verwendung von ähnlichen Wortbausteinen in Namen von Klassen und Schnittstellen. Beispiele sind `Service`, `Context` und der Name von Oberklassen als Suffix. Durch diese Namen existieren viele Paare von Elementen mit ähnlichen Namen, welche aber nicht zueinander gehören.

Wahl der Schwellenwerte der Textähnlichkeit Ein höherer Schwellenwert verringert den Anteil der unerwarteten Vorkommen. Gleichzeitig sinkt die Übereinstimmung mit dem erwarteten Modell, in einigen Fällen nähert sich diese an die Übereinstimmung mit dem anderen Modell an. Dadurch wird eine Entscheidung erschwert und eine inkorrekte Entscheidung wahrscheinlicher. Für die Suche im Architekturmodell wird für Levenshtein-Ähnlichkeit ein Schwellenwert von 0,5 verwendet. Die Idee dahinter ist, dass bei einem Wert größer oder gleich 0,5 mehr als die Hälfte des Wortes übereinstimmen. Für die Suche im Codemodell wird der Wert 0,75 verwendet, um weniger unerwartete Übereinstimmungen zu erhalten. Damit wird den ähnlichen Namen in Codemodellen entgegengewirkt. Durch die beschriebene Verwendung häufiger Wortbestandteile haben auch Bezeichner von nicht zusammenhängenden Elementen eine im Schnitt hohe Ähnlichkeit. Insgesamt schwanken die Ergebnisse für beide Fälle so sehr, dass eine eindeutige Entscheidung für einen Wert nicht möglich ist.

Dass die Namen im Diagramm sehr ähnlich mit dem Ursprungsmodell, aber stark unterschiedlich zum anderen Modell sind, ist nicht immer realistisch. Ein Beispiel dafür sind Diagramme, welche eine der Architektur folgenden Paketstruktur darstellen. Daher wird die vorherige Untersuchung mit Paketdiagrammen statt allgemeinen Codediagrammen wiederholt. Diese synthetischen Paketdiagramme werden erstellt, indem alle Elemente aus synthetischen Codediagrammen entfernt werden, die kein Paket darstellen. Der Suffix

`Impl` wird nicht mehr angehängt, stattdessen wird die Änderung *rename* auf 25% der Boxen angewendet. Die Übereinstimmung mit dem Ursprungsmodell ist dann etwa 75%, was sich direkt mit dem Änderungs-Verhältnis deckt. Für das andere Modell ist die Übereinstimmung geringer. Ist das Codemodell das Ursprungsmodell, so ist die Übereinstimmung mit dem Architekturmodell im Schnitt geringer als für den umgekehrten Fall. Das Betrachten von Paketdiagrammen löst also nicht, dass es in den synthetischen Diagrammen wenig Übereinstimmung mit beiden Diagrammen gibt, was in realen Diagrammen der Fall sein kann. Der Vorteil dieser Anpassung ist, dass die Ergebnisse weniger schwanken.

In den betrachteten Modellen ist die Paketstruktur und Architektur durchaus ähnlich. Die geringe Übereinstimmung zwischen Paketdiagrammen und Architektur ist wieder mit Namenskonvention zu erklären. Pakete werden durchgängig mit Kleinbuchstaben benannt, während Komponenten sprachunabhängig sind und daher auch Großbuchstaben enthalten können. Die bisher verwendete Levenshtein-Ähnlichkeit betrachtet Groß- und Kleinbuchstaben als unterschiedliche Zeichen, sodass die gleichen Wörter in unterschiedlichem Schema eine geringe Ähnlichkeit haben. Sollte diese Eigenschaft auch in realen Diagrammen auftreten, könnte sie von Vorteil sein. Dies vereinfacht die Typ-Erkennung, da bereits die Art der Namen eine Aussage über die Art des Modells zulässt. Gleichzeitig können dadurch aber auch Vorkommen übersehen werden, insbesondere wenn Diagramm und Modell unterschiedlicher Namenskonvention folgen.

Insgesamt zeigt sich, dass die synthetischen Diagramme in der jetzigen Form nicht geeignet sind, um systematisch die Qualität der Typ-Erkennung zu untersuchen und zu verbessern.

5.1.3 Erste Zwischenevaluation mit realen Diagrammen

Da mit synthetischen Diagrammen keine weiteren Verbesserungen für die Typ-Erkennung gefunden wurden, werden reale Diagramme betrachtet. Der Parameter `matching_threshold` wird auf 0,05 gesetzt, da geeignete Werte noch nicht bekannt sind und der geringe Wert keine Wahl ausschließt. In Tabelle 5.1 sind die Ergebnisse bezüglich der gefundenen Vorkommen dargestellt. Die Spalte *Übereinstimmung* gibt die Übereinstimmung des Diagramms mit dem gewählten Modell an und Δ die Differenz zur Übereinstimmung mit dem anderen Modell. Ist Δ negativ, so ist die Übereinstimmung mit dem im Goldstandard vorgesehene Modell geringer als mit dem anderen Modell. In dieser Zwischenevaluation wählt der Ansatz für alle Diagramme bis auf `TM_P` genau das im Goldstandard zugeordnete Modell. Für `TM_P` wählt der Ansatz statt des Codemodells das Architekturmodell. Auffällig ist, dass die Ausbeute für `BBB`, `TM_A` und `TM_P` niedrig ist. Ein möglicher Grund ist die bereits angesprochene Eigenschaft der Bezeichner bezüglich der Verwendung von Groß- und Kleinschreibung. In Code folgt dies in Java festgelegten Konventionen, in Architekturmodellen ist dies nicht der Fall. Da die Levenshtein-Distanz die Groß- und Kleinschreibung berücksichtigt, ist die Ähnlichkeit zwischen inhaltlich ähnlichen Bezeichnern ein geringer Wert.

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,62	0,46	0,95	0,56	0,70
TM_A	0,20	0,20	1,00	0,26	0,41
TM_P	0,28	-0,08	1,00	0,53	0,69
TM_U	0,32	0,24	0,80	0,86	0,83
TS	1,00	0,33	0,75	0,88	0,81
MS	0,91	0,18	0,83	0,83	0,83

Tabelle 5.1: Erste Evaluation der Typ-Erkennung; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v4`

5.1.4 Zwischenevaluation mit von Groß- und Kleinschreibung unabhängiger Levenshtein-Distanz

Die Zwischenevaluation des vorherigen Abschnittes wird mit angepasster Levenshtein-Distanz wiederholt. Die angepasste Funktion konvertiert alle Eingaben zu Kleinbuchstaben, bevor die Levenshtein-Ähnlichkeit berechnet wird. Dadurch ist die Ähnlichkeit unabhängig von Groß- und Kleinschreibung.

Mit dieser Anpassung ergeben sich die Ergebnisse in Tabelle 5.2. In dieser Zwischenevaluation wählt der Ansatz für alle Diagramme bis auf `TM_P` und `TS` das im Goldstandard zugeordnete Modell. Für `TM_P` wird weiterhin statt des Codemodells das Architekturmodell gewählt. Bei `TS` ist die Übereinstimmung mit beiden Modellen 1,0, somit wählt der Ansatz beide Modelle. Die Präzision der gefundenen Vorkommen ist für `BBB`, `TM_P` und `TM_U` gesunken. Die Ausbeute ist für `BBB`, `TM_P`, `TM_U` und `TS` gestiegen. Insgesamt ist die Qualität gemessen am F1-Score für einige Diagramme gestiegen und für `TM_U` gesunken, was eine positive Eigenschaft der Anpassung ist. Die Entscheidung für beide Modelle bei `TS` ist aus zwei Gründen keine negative Eigenschaft: Erstens erkennt der Ansatz korrekt, dass die Übereinstimmung mit dem Architekturmodell sehr hoch ist. Zweitens stimmen im `TS`-Projekt die Architektur und die Paketstruktur stark überein, weshalb eine hohe Übereinstimmung eines Architekturdiagramms mit dem Codemodell naheliegend ist. Die Entscheidung für das Codemodell bei `TM_P` ist dagegen eine negative Eigenschaft. Um diese zu beheben, werden in den folgenden Abschnitten weitere angepasste Textähnlichkeitsfunktionen und die Wahl der Parameter untersucht.

5.1.5 Wahl der Textähnlichkeitsfunktion

Der wichtigste Bestandteil der Typ-Erkennung ist der Vergleich von Texten. Diese Texte können Bezeichner von Modelelementen wie Klassen oder Komponenten sein, oder die Beschriftung von Boxen in Diagrammen. In den bisher beschriebenen Untersuchungen wurde die auf der Levenshtein-Distanz basierende Levenshtein-Ähnlichkeit verwendet. Zusätzlich zu dieser existieren weitere Funktionen zum Vergleich von Texten, wie die Jaro-Winkler- und Jaccard-Funktion, siehe Abschnitt 2.2. Im Folgenden werden erste

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,92	0,77	0,85	0,81	0,83
TM_A	0,27	0,27	1,00	0,26	0,41
TM_P	0,40	-0,12	0,84	0,58	0,69
TM_U	0,36	0,28	0,72	0,93	0,81
TS	1,00	**0**	0,77	1,00	0,87
MS	0,91	0,18	0,84	0,86	0,85

Tabelle 5.2: Typ-Erkennung mit Levenshtein ohne Berücksichtigung der Groß- und Kleinschreibung;
`similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v5`

diese weiteren Funktionen betrachtet und dann eine für den Ansatz angepasste Funktion vorgestellt.

5.1.5.1 Zwischenevaluation mit weiteren Textähnlichkeitsfunktionen

In diesem Abschnitt werden weitere übliche Textähnlichkeitsfunktionen evaluiert. Weitere Information zu den verwendeten Funktionen ist in Abschnitt 2.2 zu finden. Für diese Evaluation verwenden alle drei Funktionen zu Kleinbuchstaben konvertierte Eingaben. Die Jaccard-Funktion betrachtet jedes Zeichen als eigenes Token. In Tabelle 5.3 sind die Ergebnisse dargestellt, dabei sind die Ergebnisse für die Levenshtein-Distanz aus Tabelle 5.2 entnommen. Je Funktion ist die größere der beiden Übereinstimmungen, der F1-Score und die Wahl der Modelle, also die Klassifizierung des Diagramms, dargestellt. Das Zeichen `A` steht für das Architekturmodell und `C` für das Codemodell. Die im Goldstandard gewählte Klassifizierung ist mit einem Stern (*) markiert.

Es ist zu erkennen, das die verwendete Funktion eine Auswirkung auf F1-Score und die getroffene Klassifizierung hat. Der F1-Score ist für beide neuen Funktionen im Schnitt geringer als mit der Levenshtein-Ähnlichkeit. Die Jaro-Winkler-Ähnlichkeit wählt häufiger beide Modelle gleichzeitig. Mit der Jaccard-Ähnlichkeit ist der F1-Score im Schnitt höher als mit der Jaro-Winkler-Ähnlichkeit. Für das kommende Vorgehen wird die Jaro-Winkler-Ähnlichkeit daher nicht weiter betrachtet. Die Klassifizierung ist mit Jaccard-Ähnlichkeit die gleiche wie mit der Levenshtein-Ähnlichkeit. Die Jaccard-Ähnlichkeit wird weiter untersucht, da es möglich ist, statt einzelner Zeichen auch Wörter als Token zu betrachten. Dies biete sich an, da in beiden Modellen Bezeichner aus Wörtern bestehen, die entweder durch Leerzeichen oder Konvention (zum Beispiel `camelCase`) getrennt sind. Eine Anpassung der Gewichtung bestimmter Begriffe ist so möglich.

5.1.5.2 Angepasste Jaccard-Ähnlichkeit

Eine einfache Variante der Jaccard-Funktion nutzt Zeichen als Token. Jeder Text wird also als Menge der enthaltenen Zeichen betrachtet. Wörter, welche die gleichen Zeichen in

Diagramm	Levenshtein			Jaro-Winkler			Jaccard		
	Übst.	F1	Klass.	Übst.	F1	Klass.	Übst.	F1	Klass.
BBB	0,92	0,83	A*	1,00	0,37	A*	0,92	0,63	A*
TM_A	0,27	0,41	A*	0,87	0,51	A*	0,40	0,48	A*
TM_P	0,40	0,69	A	1,00	0,19	A, C*	0,60	0,53	A
TM_U	0,36	0,81	C*	0,76	0,16	A	0,44	0,17	C*
TS	1,00	0,87	A*, C	1,00	0,27	A*, C	1,00	0,67	A*, C
MS	0,91	0,85	A*	1,00	0,39	A*, C	0,91	0,69	A*

Tabelle 5.3: Evaluation verschiedener Textähnlichkeitsfunktion für die Typ-Erkennung; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v6`; `eval-stage1-v7`

unterschiedlicher Reihenfolge enthalten, werden dann als ähnlich gewertet. Dies scheint eine unerwünschte Eigenschaft für den gegebenen Anwendungszweck zu sein, da die Ähnlichkeit zweier Texte als Indikator der Ähnlichkeit der Bedeutung der Texte verwendet wird. Daher wird die Jaccard-Ähnlichkeit so angepasst, dass Wörter als Token verwendet werden. Ein Text wird entlang von Leerzeichen und Namenskonvention in Wörter aufgeteilt. Ein Beispiel für die Aufteilung anhand von Konvention ist die Aufteilung von `camelCalse` zu `camel` und `Case`. Alle Wörter werden in Kleinbuchstaben umgewandelt. Entsprechend der Definition der Jaccard-Funktion würde die Wörter-Liste jedes Textes als Menge betrachtet und der Schnitt der beiden Mengen durch die Vereinigung geteilt. Da die Texte der Diagramme und Modelle fehlerbehaftet sein können, führen Mengenoperationen auf Basis von Textgleichheit zu möglicherweise unerwünschten Ergebnissen. Deshalb wird eine angepasste Jaccard-Ähnlichkeit verwendet. Diese verwendet statt Mengenoperationen die Summe von Levenshtein-Ähnlichkeiten. Dazu wird die Operation *sim* eingeführt, welche die Ähnlichkeit zweier Listen a und b berechnet, welche beide aus Wörtern bestehen.

$$sim(a, b) = \sum_{w_a \in a} \sum_{w_b \in b} levenshtein(w_a, w_b) \tag{5.1}$$

Dabei ist *levenshtein* die bereits verwendete Groß- und Kleinschreibung ignorierende Levenshtein-Ähnlichkeit. Die Funktion *sim* wird statt der Schnitt-Operation auf Mengen verwendet. Somit lautet die Definition der angepassten Jaccard-Ähnlichkeit:

$$jaccard'(a, b) = \frac{sim(a, b)}{sim(a, a) + sim(b, b) - sim(a, b)} \tag{5.2}$$

Sind beide Texte maximal ein Wort lang, wird die Levenshtein-Ähnlichkeit verwendet. Die angepasste Jaccard-Ähnlichkeit unterscheidet sich von der Levenshtein-Ähnlichkeit hauptsächlich darin, dass die angepasste Funktion unabhängig von der Reihenfolge der Wörter ist. Im Folgenden werden weitere Versionen der angepassten Funktion betrachtet.

Gewichtung der Wortlänge Die erste Version der Anpassung gewichtet alle Wörter einheitlich. Dies bedeutet, dass eine kleine Änderung eines Textes eine große Auswirkung

auf die Ähnlichkeit hat, wenn die Änderung in einem kurzen Wort stattfindet. Dadurch kann zum Beispiel ein Prefix wie `I`, welcher als einzelnes Wort gewertet wird, eine geringe Ähnlichkeit verursachen, sollte der Präfix in nur einem der beiden Bezeichner vorkommen. Um diese Eigenschaft abzuschwächen, werden alle Wörter mit einem Gewicht versehen. Das Gewicht ω_{w_x} eines Wortes w_x ist dabei die relative Länge des Wortes im Text X des Wortes, dargestellt in Gleichung 5.4. Für jedes Wort in einem Text X wird also die Länge des Wortes durch die Länge des längsten Wortes in diesem Text geteilt. Somit ergibt sich folgende neue Definition der Funktion *sim*:

$$sim(a,b) = \sum_{w_a \in a} \sum_{w_b \in b} \omega_{w_a} \cdot \omega_{w_b} \cdot levenshtein(w_a, w_b) \tag{5.3}$$

$$\omega_{w_x} = \frac{|w_x|}{\max_{w_\chi \in X} |w_\chi|} \tag{5.4}$$

Gewichtung häufiger Begriffe aus der Softwaretechnik In Unterabschnitt 5.1.2 wird beschrieben, dass Bezeichner in Codemodellen ähnliche Begriffe enthalten. Somit ist die Ähnlichkeit verschiedener Bezeichner hoch, obwohl diese unabhängige Bestandteile beschreiben. Anslow u. a. [2] befassen sich mit in Bezeichnern vorkommenden Begriffen. Es werden die Häufigkeit verschiedener Begriffe in Bezeichnern in Java-Programmen gezählt. Am häufigsten kommen die Begriffe `Test`, `Action`, `Impl`, `Factory`, `Exception` und `Data` vor.

Diese Begriffe folgen Konvention aus Sicht der Softwareentwicklung, nicht aus Sicht der Domäne. So beschreibt zum Beispiel `Exception`, dass eine Klasse eine Ausnahme darstellt, aber nicht in welchem Teil eines Projekts diese verwendet wird. Die Suche nach übereinstimmenden Elementen zwischen Diagrammen und Modellen soll dagegen aus Sicht der Domäne erfolgen.

Um dies für die Ähnlichkeit zu berücksichtigen, wird die Ähnlichkeitsfunktion angepasst, sodass diese häufigen Begriffe geringer gewichtet werden. Dazu wird für jedes Wort x ein Gewicht λ_{w_x} berechnet, welches für Wörter in der Menge der häufigen Begriffe 0,25 und für alle anderen Wörter 1,0 ist. Die angepasste Definition der Funktion *sim* lautet dann:

$$sim(a,b) = \sum_{w_a \in a} \sum_{w_b \in b} \omega_{w_a} \cdot \omega_{w_b} \cdot \lambda_{w_a} \cdot \lambda_{w_b} \cdot levenshtein(w_a, w_b) \tag{5.5}$$

Eine ähnliche Berücksichtigung dieser häufigen Begriffe wird von Telge [27] verwendet. Die Arbeit verwendet eine Heuristik, welche nach Bezeichnern sucht, die sich nur um häufige Begriffe unterscheiden.

Gewichtung häufiger Projekt-Begriffe Eine Alternative zur Gewichtung von Begriffen aus der Softwareentwicklung ist die Gewichtung von Begriffen, welche häufig im Projekt vorkommen. Die Idee dahinter ist, dass häufig verwendete Begriffe weniger Aussagekraft über die Ähnlichkeit zweier Bezeichner haben.

Um diese Gewichtung durchzuführen, werden die Vorkommen aller Wörter in den Bezeichnern gezählt und dann für jedes Wort die relative Häufigkeit berechnet. Die relative

Häufigkeit eines Wortes w_x ist dabei die Anzahl der Vorkommen eines Wortes, geteilt durch die Anzahl des häufigsten Wortes. Somit ergibt sich für jedes Wort eine Zahl zwischen null und eins, wobei eine höhere Zahl ein höheres Vorkommen angibt. Da die Gewichtung die Bedeutung seltener Wörter erhöhen soll, wird der Wertebereich umgedreht. Zusätzlich wird der Wertebereich skaliert, sodass alle Gewichte über einem Mindestwert liegen.

5.1.5.3 Zwischenevaluation der angepassten Jaccard-Funktion

In diesem Abschnitt werden die eben dargestellten Anpassungen der Jaccard-Funktion evaluiert.

Die erste Anpassung ist die Verwendung von Wörtern statt einzelner Zeichen als Token sowie die Verwendung der Levenshtein-Ähnlichkeit für Mengenoperationen. Die Ergebnisse sind in Tabelle 5.4 dargestellt. Für alle Diagramme bis auf `TM_P` wird genaue das im Goldstandard zugeordnete Modell gewählt. In keinem Fall werden beide Modelle gewählt, bei `TS` wird das Architekturmodell anstelle des Codemodells gewählt. Verglichen mit den Ergebnissen bei Verwendungun der Levenshtein-Ähnlichkeit ist der F1-Score für alle Diagramme bis auf `TM_U` gesunken. Im Vergleich zu der originalen Jaccard-Funktion erzielt diese Version im Schnitt einen höheren F1-Score. Da im Vergleich zu den vorherigen Ähnlichkeits-Funktionen aber keine F1-Scores unter 0,3 auftreten, wird diese angepasste Funktion weiter untersucht.

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
`BBB`	0,77	0,62	0,96	0,64	0,77
`TM_A`	0,20	0,13	1,00	0,17	0,30
`TM_P`	0,80	-0,40	0,44	0,58	0,50
`TM_U`	0,40	0,24	0,72	0,93	0,81
`TS`	1,00	0,16	0,83	0,88	0,86
`MS`	0,73	0,45	1,00	0,50	0,67

Tabelle 5.4: Evaluation der Typ-Erkennung mit angepasster Jaccard-Funktion; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v14`

Die zweite Anpassung beinhaltet die Gewichtung der Wörter abhängig von ihrer relativen Länge im Ursprungs-Bezeichner. Eine Beschreibung ist in Absatz 5.1.5.2 zu finden. Die Ergebnisse sind in Tabelle 5.5 dargestellt und werden mit der vorherigen Version verglichen. Für alle Diagramme ist die Ausbeute gestiegen, die Präzision hat sich für alle Diagramme bis auf `TS` verbessert. Daher ist der F1-Score für alle Diagramme bis auf `TS` gestiegen. Der Ansatz entscheidet sich für alle Diagramme bis auf `TM_P` für das im Goldstandard zugeordnete Modell. Für `TM_P` wird weiterhin statt des Codemodells das Architekturmodell gewählt, der Abstand zwischen beiden Übereinstimmungen ist gesunken. Bei Diagramm `MS` ist der Abstand zwischen den Übereinstimmungen stark gesunken.

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,92	0,62	0,79	0,83	0,81
TM_A	0,40	0,33	1,00	0,48	0,65
TM_P	0,68	-0,12	0,51	0,72	0,60
TM_U	0,48	0,40	0,74	1,00	0,85
TS	1,00	0,17	0,57	0,94	0,71
MS	0,82	0,09	0,68	0,78	0,73

Tabelle 5.5: Evaluation der Typ-Erkennung mit angepasster Jaccard-Funktion und Gewichtung langer Wörter; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v15`

Die dritte Anpassung beinhaltet, dass in der Softwareentwicklung häufig verwendete Begriffe geringer gewichtet werden. Eine Beschreibung ist in Absatz 5.1.5.2 zu finden. Die Ergebnisse sind in Tabelle 5.6 dargestellt. Es ist zu erkennen, dass die Anpassung für alle Diagramme bis auf TM_U eine geringe Auswirkung auf den F1-Score hat. Für TM_U ist die Präzision und somit der F1-Score gesunken. Der Ansatz entscheidet sich für alle Diagramme bis auf TM_P für das im Goldstandard zugeordnete Modell. Für TM_P wird weiterhin statt des Codemodells das Architekturmodell gewählt, der Abstand zwischen beiden Übereinstimmungen ist weiter gesunken. Insgesamt bringt diese Anpassung für die meisten Diagramme nur einen sehr geringen Vorteil und für TM_U einen Nachteil. Daher wird diese Anpassung nicht verwendet.

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,92	0,62	0,78	0,83	0,81
TM_A	0,40	0,33	1,00	0,48	0,65
TM_P	0,60	-0,08	0,50	0,72	0,59
TM_U	0,52	0,44	0,63	1,00	0,77
TS	1,00	0,16	0,55	0,94	0,69
MS	0,82	0,09	0,69	0,81	0,74

Tabelle 5.6: Evaluation der Typ-Erkennung mit angepasster Jaccard-Funktion und Gewichtung von Begriffen aus der Softwareentwicklung; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v16`

In der vierten Anpassung wird statt einer Gewichtung häufiger Begriffe der Softwareentwicklung eine Gewichtung nach Häufigkeit in den Modellen durchgeführt. Die entsprechende Beschreibung ist in Absatz 5.1.5.2 zu finden. Tabelle 5.7 zeigt die Ergebnisse. Die Übereinstimmung ist für alle Diagramme die gleiche wie in der zweiten Anpassung. Entsprechend ist auch die Wahl des Ansatzes gleich. Der durchschnittliche F1-Score hat sich leicht verbessert, dabei hat sich der konkrete Wert für einige der Diagramme verbessert

und für andere verschlechtert. Für das weitere Vorgehen wird diese Anpassung anstelle der dritten Anpassung verwendet, da insgesamt eine Verbesserung erzielt wird.

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,92	0,61	0,89	0,89	0,89
TM_A	0,40	0,33	1,00	0,48	0,65
TM_P	0,68	-0,12	0,54	0,75	0,63
TM_U	0,48	0,40	0,67	1,00	0,80
TS	1,00	0,17	0,70	0,94	0,80
MS	0,82	0,09	0,61	0,75	0,67

Tabelle 5.7: Evaluation der Typ-Erkennung mit angepasster Jaccard-Funktion und Gewichtung von häufigen Begriffen in den Projekten; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v17`

5.2 Verbindungssuche

Der zweite Schritt hat die Aufgabe, Paare von Diagramm-Boxen und Modell-Elementen zu finden, welche einander repräsentieren. Es werden also Nachverfolgbarkeitsverbindungen gesucht, der Schritt führt *trace link recovery* durch. Dabei wird die Suche auf die Modelle eingeschränkt, welche im ersten Schritt als Typ des Diagramms erkannt wurden.

Eine einfache Möglichkeit, Verbindungen zu finden, ist die Suche nach Elementen mit gleichen oder ähnlichen Namen. Dadurch können aber Inkonsistenzen wie Umbenennungen nicht als solche erkannt werden. Dies ist ein Abbildung 5.3 dargestellt. Die Abbildung zeigt zweimal dieselbe Struktur, einmal stellt diese das Diagramm da und im anderen Fall das Modell. Da alle bis auf zwei Knoten jeweils paarweise den gleichen Namen haben, ist es einfach, diese zu verbinden. Aber für *C* beziehungsweise *renamed* ist die Verbindung nur aus der Struktur des Graphen ersichtlich.

Um die gesamte Struktur von Modell und Diagramm zu berücksichtigen, wird *graph matching* angewendet. Dabei werden zwei Graphen miteinander verbunden. Der erste Graph ist der Diagrammgraph, welcher aus der Boxen-und-Linien-Form des Diagramms erstellt wird. Der zweite Graph ist der Modellgraph. Dieser muss erst aus den für die Diagrammtypen relevanten Modellen extrahiert werden. Es wird also für beide Modelle eine Transformation zu einem vereinfachten Graph-Modell durchgeführt.

Für die Verbindungssuche wird in allen im Folgenden beschriebenen Untersuchungen und Zwischenevaluationen das originale Codemodell verwendet. Dieses ist in Unterabschnitt 2.4.2 beschrieben. Sollte das erweiterte Codemodell verwendet werden, so wird dies explizit erwähnt.

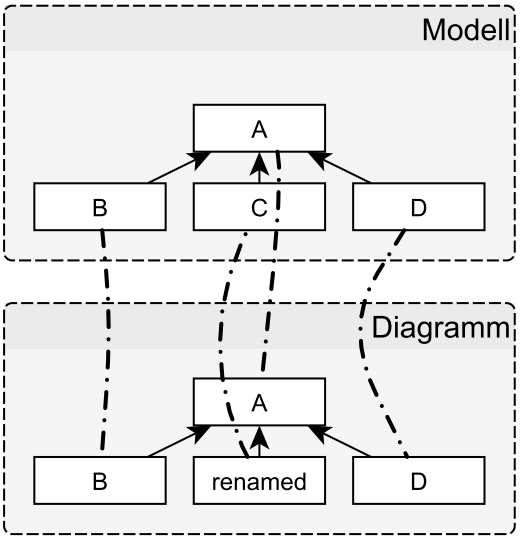

Abbildung 5.3: Verbindungen zwischen Diagramm und Modell, die meisten der Verbindungen können anhand des Textes gefunden werden

5.2.1 Anwendung des Algorithmus

Für das Diagramm und alle von der Typ-Erkennung ausgewählten Modelle müssen Graphen erstellt werden. Die Transformation von Diagramm und Modell zu Graph ist in Abschnitt 2.8 dargestellt. Sobald diese durchgeführt ist, werden beide Graphen an Similarity Flooding übergeben.

Die Implementierung von Similarity Flooding folgt der Beschreibung in Abschnitt 2.7. Dabei wird der *pairwise connectivity graph* $PCG(A, B)$ nicht explizit erstellt. Stattdessen wird direkt der *propagation graph* $PG(A, B)$ erstellt und $PCG(A, B)$ implizit verwendet. Das ist möglich, da der Algorithmus zur Erstellung des *pairwise connectivity graph* und des *propagation graph* die gleiche Iteration über alle Knotenpaare nutzen. Innerhalb dieser Iterationen können alle Schritte zur Erstellung des *pairwise connectivity graph* durchgeführt und die Ergebnisse direkt genutzt werden, ohne den Graphen explizit zu erstellen.

Ein wichtiger Bestandteil des Algorithmus sind Zuordnungen. Diese sind eine Abbildung von einem Knotenpaar auf eine Ähnlichkeit. Um die Performance zu erhöhen, werden in der Implementierung alle Knotenpaare durch natürliche Zahlen repräsentiert, statt explizite Datentypen zu nutzen. Dazu wird eine Reihenfolge der Knotenpaare definiert, jedes Paar ist dann durch einen Index eindeutig identifiziert. Der Vorteil davon ist, dass die Zuordnung als eine Liste von Ähnlichkeiten implementiert werden kann und keine Datenstrukturen wie Zuordnungstabellen genutzt werden. Zusätzlich profitiert verwendete Datenstrukturen der Graphen von der einfachen Berechnung von Streuwerten von Indizes.

Die erste Zuordnung, welche die initiale Ähnlichkeit enthält, wird mit der Textähnlichkeit der Texte der Knoten berechnet, zum Beispiel über die Levenshtein-Ähnlichkeit. Die

Ähnlichkeit wird für alle möglichen Knotenpaare, also für das kartesische Produkt der Menge der Boxen und der Menge der Modellelemente, vorberechnet.

Durch das Einführen eines Schwellenwertes für die Textähnlichkeit können unpassende Verbindungen minimiert werden. Die Ähnlichkeit aller Paare mit einem Wert unter dem Schwellenwert wird auf 0 gesetzt. Der Schwellenwert wird durch den Parameter `text_similarity_threshold` bestimmt. Diese Paare können dann bei der Anwendung des Algorithmus Similarity Flooding keine Ähnlichkeit beisteuern. Angewendet auf synthetische Diagramme steigt so die Qualität der Verbindungssuche bei *rename*-Änderungen.

Die gefundenen ähnlichen Knoten werden gefiltert, um die Verbindungen daraus zu extrahieren.

5.2.2 Filtern der Ähnlichkeiten

Das Ergebnis des Similarity Flooding-Algorithmus ist eine Zuordnung σ zwischen den Knoten der Graphen A und B. Diese gibt für jedes Paar von Knoten (x_A, x_B) eine Ähnlichkeit zwischen 0 und 1 an. Um die zu akzeptierenden Verbindungen zu finden, werden alle Paare in der Zuordnung entsprechend ihrer Ähnlichkeit sortiert. Für jedes Paar (x_A, x_B) wird dann in absteigender Reihenfolge überprüft, ob x_A oder x_A bereits Teil einer Verbindung sind. Ist das nicht der Fall, wird eine Verbindung zwischen x_A und x_B erstellt.

Eine Schwäche des Algorithmus Similarity Flooding ist, dass dieser Knoten ohne Kanten nicht beachtet. Somit gibt es in der Ähnlichkeiten-Zuordnung keine Werte für diese Knoten. Um dies zu beheben, wird nach der Anwendung des Algorithmus und dem Filtern ein Nachbearbeitungsschritt eingeführt. Für jeden Knoten, der durch den Algorithmus keiner Verbindung zugeordnet wurde, wird eine Verbindung auf Basis der initialen Ähnlichkeit gesucht. Der Ablauf ist dabei der gleiche wie beim Filtern. Da während dem Filtern Informationen verfügbar sind, welche es ermöglichen, diesen Schritt zu beschleunigen, ist die tatsächliche Implementierung in das Filtern integriert. Durch diese Anpassung steigt die Ausbeute, insbesondere bei synthetischen Codediagrammen.

Verbindungs-Schwellenwert In der vorangegangenen Implementierung des Filterns wird für jedes Element, sofern möglich, eine Verbindung erstellt. Diagramm-Elemente werden nur dann nicht Teil einer Verbindung sein, wenn während des Verbindens keine unverbundenen Modell-Elemente mehr übrig sind. Für Modell-Elemente gilt das entsprechend umgekehrt. Somit werden alle Elemente der Seite mit weniger Elementen Teil einer Verbindung sein. Diese Eigenschaft kann die Verbindungssuche im Falle von Inkonsistenzen beeinträchtigen. Enthält ein Diagramm weniger Boxen, als es Elemente im Modell gibt, so ist der Ansatz gezwungen, jede Box einer Verbindung zuzuteilen, selbst wenn eine der Boxen eigentlich keine Entsprechung im Modell hat. Bei Betrachtung realer Diagramme ist erkennbar, dass solche Boxen ohne Entsprechung im Modell existieren. Beispiele dafür sind Boxen, welche als Platzhalter für externe Systeme stehen. Um diese Eigenschaft zu beheben, wird das Filtern so angepasst, dass Verbindungen nur dann erstellt

werden, wenn die Ähnlichkeit über einem Schwellwert liegt. Der Schwellwert für Verbindungen auf Basis der Ähnlichkeit aus dem Algorithmus ist `similarity_threshold`. Für Verbindungen auf Basis der initialen Ähnlichkeit wird der Wert des Parameters `text_similarity_threshold` verwendet.

5.2.3 Anwendung auf synthetische Diagramme

Im Folgenden wird die Auswirkung verschiedener Änderungsklassen auf den Schritt der Verbindungssuche untersucht. Die in Unterabschnitt 4.2.2 definierten Änderungen werden sowohl einzeln als auch kombiniert auf synthetische Diagramme angewendet. Dabei werden die erwarteten Verbindungen entsprechend angepasst und die generierten Diagramme an die Verbindungssuche übergeben.

Um die Menge der angewendeten Änderungen zu beschreiben, wird das Änderungs-Verhältnis eingeführt. Das Änderungs-Verhältnis beschreibt das Verhältnis der Anzahl von angewendeten Änderungen zur Anzahl der Boxen im ursprünglichen Diagramm.

Untersuchung: Alle Änderungsklassen Für die Folgende Untersuchung werden alle Änderungen gemeinsam angewendet. Das modifizierte synthetische Diagramm wird mit dem ursprünglichen Modell an die Verbindungssuche übergeben. Die Ergebnisse sind für diese Konfiguration die folgenden: Ist das Änderungs-Verhältnis 0%, so findet der Ansatz genau die vorhandenen und keine weiteren Verbindungen. Bei steigendem Änderungs-Verhältnis nimmt die Qualität der Verbindungssuche, gemessen am F1-Score, ab. Dabei sinkt die Qualität stärker als das Änderungs-Verhältnis zunimmt.

Untersuchung: Einzelne Änderungsklassen In dieser Untersuchung wird der Einfluss der einzelnen Änderungsklassen betrachtet. Dazu werden synthetische Diagramme an die Pipeline übergeben, auf die nur eine Art von Änderung mehrfach angewendet wird. Je Änderungsklasse ist die Anzahl der Anwendungen gleich und entspricht dem Änderungs-Verhältnis. Als `epsilon` wird für diese erste Untersuchung 0,5 verwendet.

Für Architekturdiagramme werden für alle Änderungsklassen sehr hohe F1-Werte um 0,95 erreicht, die nahe beieinander liegen. Da die betrachteten Architekturmodelle keine Hierarchie aufweisen, entartet *move* zu *connect*. Wegen des hohen `epsilon` kombiniert mit kleinen Diagrammen ist es möglich, dass es zu wenigen Iterationen des Algorithmus kam. Hinzugefügte Kanten konnten also nicht stark in die Berechnung einfließen. Gleichzeitig wurden die Änderungen der Namen dennoch erkannt.

Für Codediagramme ist die Qualität der Verbindungssuche, gemessen am F1-Score, im Vergleich schlechter. Codediagramme sind in der Regel größer als Architekturdiagramme und weisen ausgeprägte eine Hierarchie auf. Die Unterschiede des Einflusses der Änderungsklassen auf die Qualität ist hier ausgeprägter. Den geringsten Einfluss haben *create* und *connect*, hinzugefügte Kanten scheinen nur einen geringen Einfluss auf Similarity Flooding zu haben. Zusätzlich ist der Einfluss von *disconnect* gering. Dies lässt sich mit der Hierarchie im Codemodell erklären. Da *disconnect* nur *default*-Kanten entfernt und der

Großteil der Knoten in einer Hierarchie ist, entstehen keine unverbundenen Knoten. Die Änderung *move* hat einen größeren Einfluss als bei Architekturdiagrammen. Da Hierarchie im Modell existiert, entartet *move* seltener zu *connect*. Den größten Einfluss haben *delete* und *rename*. Für *delete* ist der Grund der, dass diese Änderung sowohl Knoten als auch Kanten entfernt und somit die größte Veränderung der Graphen bewirkt.

Wird die initiale Ähnlichkeit während dem Filtern nicht genutzt, um Verbindungen für Knoten ohne Kanten zu finden, so haben *disconnect* und *delete* einen starken negativen Effekt auf die Qualität. Der Grund dafür ist, dass diese Änderungen zu unverbundenen Knoten führen können.

5.2.4 Vergleich mit Verbindungssuche ausschließlich auf Textbasis

Wie bereits dargestellt, wird die initiale Ähnlichkeit auf Basis der Texte berechnet. Mit jeder Iteration wird die Zuordnung und die darin enthaltene Ähnlichkeit verändert. Somit ist eine relative Schwächung der Information der initialen Ähnlichkeit möglich. Die Qualität der Verbindungssuche ist stark für destruktive Strukturänderungen anfällig. Diese Änderungen, also *delete*, *disconnect* und teils auch *move*, haben keinen Einfluss auf die Textähnlichkeit. Daher soll untersucht werden, ob der Algorithmus einen Vorteil gegenüber einer Verbindungssuche ausschließlich auf Textbasis hat. Dazu wird der Algorithmus für synthetische Diagramme einmal mit maximal einer Iteration und einmal mit null Iterationen ausgeführt. Bei einer Ausführung mit null Iterationen wird direkt die initiale Ähnlichkeit verwendet. Die Qualität der Verbindungssuche, gemessen am F1-Score, ist für die Ausführung mit einer Iteration für die meisten Änderungsklassen geringfügig niedriger als für null Iterationen. Insbesondere für die Strukturänderung *move* ist die Qualität mit einer Iteration geringer. Im Falle von *rename* steigt die Qualität bei einer Iteration stark an.

Werden nur Ausschnitte der Diagramme betrachtet, so ist die Qualität für eine und keine Iteration geringer und der Abstand zwischen den beiden Werten ist größer. Gleichzeitig profitiert zusätzlich zu *rename* auch *delete* von einer Iteration.

Der Algorithmus Similarity Flooding hat also einen Vorteil gegenüber einer Verbindungssuche ausschließlich auf Textbasis, sofern die meisten Änderungen Namen und nicht Struktur betreffen. Werden nur Ausschnitte betrachtet, so hat der Algorithmus weiter einen Vorteil in den beschriebenen Fälle. Der Nachteil bei Strukturänderungen wie *move* ist aber bei Betrachtung von Auschnitten verstärkt.

5.2.5 Wahl der Parameter für synthetische Diagramme

Die Verbindungssuche beziehungsweise deren Bestandteile Similarity Flooding sowie Filtern werden durch Parameter kontrolliert. Wie bereits in Unterabschnitt 4.2.3 beschrieben, kann die Auswirkung der realistischeren synthetischen Diagramme auf die Qualität durch passende Parameter teilweise ausgeglichen werden. In diesem Abschnitt wird die

Wahl der Parameter untersucht und dargestellt. Ziel bei der Wahl der Parameter ist hauptsächlich einen hohen durchschnittlichen F1-Score zu erreichen, daher wird dieser im Text auch als Qualität bezeichnet. Zusätzlich wird auch der minimale F1-Score betrachtet, da der Ansatz für kein Diagramm nicht anwendbar sein soll.

epsilon In Unterabschnitt 2.7.3 wird bereits die Auswirkung des `epsilon`-Parameters auf das Verhalten von Similarity Flooding betrachtet. Bei den bereits beschriebenen Untersuchungen mit synthetischen Diagrammen war zu erkennen, dass die Wahl von `epsilon` abhängig von der Änderungsklasse verschiedene Auswrikungen hat. Um einen geeigneten Wert zu finden, wird die Verbindungssuche mit synthetischen Diagrammen und verschiedenen Werten für `epsilon` ausgeführt. Dies wird für jede Änderungsklasse einzeln für `epsilon`-Werte im Bereich von 0,25 bis 2,0 durchgeführt. Bei Architekturdiagrammen ist kein starker Unterschied zwischen den Werten zu erkennen. Die insgesamt hohe erreichte Qualität bei synthetischen Architekturdiagrammen und die geringe Anzahl an Elementen in diesen lässt wenig Spielraum, was kleine Änderungen in der Qualität verhindert. Die vorhandenen Qualitätsunterschiede könne hier auch mit den durch die zufälligen Änderungen entstehender Schwankungen erklärt werden. Bei Codediagrammen kann dagegen eine Abhängigkeit von der Änderungsklasse erkannt werden. Insbesondere für *rename* scheint ein kleineres `epsilon` besser zu sein. Für andere Änderungen scheinen sowohl höhere (1,5) als auch niedrigere Werte (0,5) besser zu sein, mit schlechten Ergebnissen dazwischen. Die Wahl eines Wertes für `epsilon` wird durch dieses Minimum um 1,0 erschwert, insbesondere da je nach Änderungsklasse eine andere Richtung aus dem Minimum besser ist. Da es keinen eindeutig besseren Wert gibt und 1,0 an den für jede Änderungsklasse optimalen Werten ist, wird 1,0 als Standardwert verwendet. Es zeigt sich also, dass der geeignete Wert abhängig von den im Diagramm vorhandenen Inkonsistenzen ist.

Formeln Die Auswahl der Fixpunkt- und Koeffizienten-Formel hat auch eine Auswirkung auf die Qualität der Verbindungssuche. Die in der Arbeit zu Similarity Flooding [21] empfohlenen Formeln erreichen für die meisten Änderungsklassen eine höhere Qualität. Für *rename* verschlechtert sich die Qualität mit der Fixpunktformel C jedoch. Ein möglicher Grund dafür ist, dass die Formeln die initiale Ähnlichkeit, welche auf den Namen der Boxen basiert, stärker berücksichtigt. *rename* ist eine Namensänderung und wirkt sich damit direkt auf die initiale Ähnlichkeit aus. Um eine hohe Ähnlichkeit nach einer Umbenennung zu erreichen, sind mehrere Iterationen des *graph matching* nötig. Wird in diesen die initiale Ähnlichkeit stärker berücksichtigt, in welcher die Umbenennungen durch geringe Ähnlichkeit der betroffenen Elemente dargestellt ist, kann dies die Anzahl nötiger Iterationen erhöhen. Die anderen Änderungen sind Strukturänderungen, welche sich nicht oder nur wenig auf die initiale Ähnlichkeit auswirken.

Schwellenwert Der Schwellenwert für die Textähnlichkeit, also der Parameter `text_similarity_threshold`, hat auch einen Einfluss auf die Qualität der Verbindungssuche. Für die Änderungen *delete*, *move*, und insbesondere *rename*, steigt die Qualität mit einem höheren Schwellenwert. Für einige der Änderungen verändert sich die Qualität

nicht. Überprüft wurden Werte zwischen 0,0 und 0,5 in Schritten von 0,1. Der Wert 0,5 ist in dem Bereich der beste Wert und wird daher gewählt. In Fällen, in denen nur ein Ausschnitt des kompletten synthetischen Diagramms genutzt wird, ist in den meisten Fällen weiterhin 0,5 der beste Wert. Für reale Anwendungsfälle mit mehr Unterschieden in den Namen ist potentiell ein niedrigerer Wert oder ein relativer Wert besser geeignet.

Werden synthetische Diagramme mit gemischten Änderungsklassen betrachtet und die beschriebenen Parameter und Formeln verwendet, so ist die Qualität weniger abhängig vom `epsilon`. Dabei ist zu beachten, dass die Anteile der Änderungen an der Mischung gleichverteilt zufällig gewählt ist. Eine solche Gleichverteilung ist in realen Anwendungsfällen nicht zu erwarten. Zusätzlich haben alle Änderungstypen bis auf *rename* eine größere Auswirkung auf die Struktur als auf die Namen. Somit ist die Mischung größtenteils mit Änderungen gefüllt, für die der Algorithmus nicht primär ausgelegt ist. Dies erklärt, warum ein höherer Wert die Qualität nicht senkt, obwohl dies für *rename* von Nachteil ist. Die Qualität bei einem niedrigen Wert kann damit erklärt werden, dass die Anpassungen und gewählten Parameter, insbesondere die Fixpunktgleichung, den Informationsverlust durch viele Iterationen verhindern. Der Wert 1,0 wird als Standardwert beibehalten. Dieser Wert erreicht eine akzeptable Qualität für *rename*, welches im Fokus der Arbeit steht. Gleichzeitig ist der Wert für die anderen Änderungen sowie das Laufzeitverhalten akzeptabel. Insgesamt ist die Relevanz von `epsilon` dank der Wahl geeigneter Parameter und Formeln geringer.

5.2.6 Zwischenevaluation mit realen Diagrammen

Da auf synthetischen Diagrammen eine hohe Qualität erzielt werden kann, wird in diesem Abschnitt überprüft, wie gut der Ansatz auf reale Diagramme übertragen werden kann. Der Parameter `similarity_threshold` wird auf 0,1 gesetzt. Sollte die berechnete Ähnlichkeit zweier Elemente unter diesem Schwellwert liegen, wird keine Verbindung zwischen den Elementen erstellt. Die Ergebnisse sind in Tabelle 5.8 dargestellt. Die Qualität der Ergebnisse für alle Diagramme über 0,70. Für `TM_U` lässt sich die geringere Präzision damit erklären, dass im Diagramm viele Boxen keine Entsprechung im Modell haben. Der Ansatz findet trotz des Schwellenwertes für viele dieser Boxen eine Verbindung, die nicht erwartet wird. Ein größerer Schwellenwert könnte für `TM_U` also geeignet sein. Ist der Wert niedriger oder es wird kein Schwellenwert verwendet, so sinkt die Präzision für Diagramme mit Boxen ohne Entpsrechung im Modell stark ab. Für `TM_U` wird dann eine Präzision von unter 0,40 erreicht, da der Ansatz für jede Box im Diagramm eine Verbindung erstellt.

Im Vergleich zur Qualität bei den Untersuchungen mit synthetischen Diagrammen ist die Qualität hier geringer. Ein möglicher Grund dafür ist, dass die realen Diagramme mehr Inkonsistenzen enthalten. Dafür spricht, dass die Qualität bei `MS` am höchsten ist. Das Diagramm `MS` ist womöglich generiert und folgt einer formalen Spezifikation.

Diagramm	Präzision	Ausbeute	F1
BBB	1,00	0,75	0,86
TM_A	1,00	0,75	0,86
TM_P	0,83	0,86	0,84
TM_U	0,66	1,00	0,80
TS	0,80	0,67	0,73
MS	0,91	1,00	0,95

Tabelle 5.8: Zwischenevaluation der Verbindungssuche; epsilon: 1,0; max_iterations: 100; text_similarity_threshold: 0,5; similarity_threshold: 0,1; git-tag: eval-stage2-v22

5.3 Inkonsistenz-Erkennung

Sobald die Verbindungen hergestellt sind, kann nach Inkonsistenzen gesucht werden. Zur Erkennung von Inkonsistenzen werden definierte Regeln verwendet. Für jeden gewählten Modell-Typ werden die Regeln auf alle Verbindungen angewendet. Zusätzlich werden die Regeln auch auf alle Boxen und Modellelemente angewendet, welche nicht Teil einer Verbindung sind. Abhängig vom gewählten Modell-Typ wird eine unterschiedliche Menge von Regeln verwendet.

Für die Inkonsistenz-Erkennung wird in allen im folgenden beschriebenen Untersuchungen und Zwischenevaluationen das originale Codemodell verwendet. Dieses ist in Unterabschnitt 2.4.2 beschrieben. Sollte das erweiterte Codemodell verwendet werden, so wird dies explizit erwähnt.

5.3.1 Konsistenz-Regeln

Die Regeln sind auf Basis der in Abschnitt 2.6 beschriebenen Inkonsistenzen definiert. Für alle Inkonsistenzen bis auf `missing_box` wird von den Regeln eine vollständige Übereinstimmung zwischen Diagramm und Modell erwartet. Für die Inkonsistenz `missing_box` werden spezifischere Regeln verwendet. Es ist anzumerken, dass die Nichtbeachtung einer Regel nicht bedeutet, dass ein Diagramm fehlerhaft ist. Bewusste Entscheidungen von Entwicklern bei der Erstellung von Diagrammen können zu scheinbaren Inkonsistenzen führen, zum Beispiel wenn in Modell und Diagramm unterschiedliche Konventionen verwendet werden. Aufgrund der informellen Natur der Diagramme gibt es keine Spezifikation, welche Inhalte eingebaut oder ausgelassen werden müssen. Daher sind gefundene Inkonsistenzen im Allgemeinen nicht als Fehler, sondern als Hinweise zu verstehen. Die verwendeten Regeln sind:

1. `SameNameForLinkedElements`: Die Namen zweier verbundener Elemente sollten gleich sein. Ist dies nicht der Fall, so ist dies eine `name_inconsistency`.
2. `AllBoxesMustBeLinked`: Für jede Box wird erwartet, dass diese Teil einer Verbindung ist. Ist dies nicht der Fall, so ist dies eine Inkonsistenz der Art `unexpected_box`. Diese

Inkonsistenz kann sowohl darauf hindeuten, dass eine Box im Diagramm etwas darstellt, was nicht (mehr) im Modell existiert, als auch dass keine Verbindung zum tatsächlich dargestellten Modellelement gefunden wurde. Es ist auch möglich, dass die Box im Diagramm nicht zur Darstellung eines Modellelements, sondern zum Beispiel zur Verbesserung der Visualisierung genutzt wird.

3. `AllModelEntitiesMustBeRepresented`: Diese Regel erwartet, dass jedes Modellelement im Diagramm dargestellt wird. Dies ist gegeben, wenn jedes Modellelement Teil einer Verbindung ist. Ist ein Modellelement nicht Teil einer Verbindung, so ist dies eine Inkonsistenz der Art `missing_box`. Für große Modelle, wie es Codemodelle sind, ist es nicht realistisch, das gesamte Modell in einem Diagramm darzustellen. Daher wird diese Regel nur für Architekturdiagramme verwendet.

4. `EntitiesMustBeConnectedExactlyToDependencies`: Für jede Box wird erwartet, dass genau zu jeder Abhängigkeit eine Linie und zu allen anderen Knoten keine Linie existiert. Dabei sind nur Elemente, welche im Diagramm dargestellt werden, die also Teil einer Verbindung sind, relevant. Das Vorhandensein aller Abhängigkeiten eines Elements wird von dieser Regel nicht verlangt. Eine Abhängigkeit ist jede Beziehung, welche in Graphen mit einer *default*-Kante dargestellt wird. Fehlt die Linie, so ist dies eine Inkonsistenz der Art `missing_line`. Besteht eine Linie zu einer Box, welche keine Abhängigkeit darstellt, so ist dies eine Inkonsistenz der Art `unexpected_line`.

5. `BoxesMustBeInParent`: Die Regel erwartet, dass eine Box im Diagramm in der Box enthalten ist, die in der Modell-Hierarchie direkt über ihr ist. Diese Erwartung besteht nur, wenn die erwartete beinhaltende Box im Diagramm vorhanden ist. Sollte eine Box im Diagramm nicht in der erwarteten Box enthalten sein, so ist dies eine `hierarchy_inconsistency`. Falls eine Box im Diagramm Kind einer anderen Box ist und dies im Modell nicht der Fall ist, so ist dies auch eine `hierarchy_inconsistency`. Diese Regel wird nicht angewendet, wenn die beinhaltende Box im Diagramm selbst nicht Teil einer Verbindung, also unerwartet, ist.

6. `PackagesMustContainAllSubpackagesIfOneIsEmpty`: Diese Regel wird nur für Codediagramme verwendet. Für alle Boxen, welche ein Paket darstellen, wird erwartet, dass Folgendes gilt: Enthält die Box ein Unterpaket, welches ohne Inhalt dargestellt ist, so müssen alle anderen Unterpakete auch in der Paket-Box enthalten sein. Ist dies nicht der Fall, so ist dies eine Inkonsistenz der Art `missing_box`.

Für alle Regeln gilt, dass Inkonsistenzen im Zusammenhang mit unerwarteten Boxen ignoriert werden. Dadurch wird vermieden, dass zum Beispiel unerwartete Kanten ausgehend von einer unerwarteten Box gemeldet werden, da die unerwarteten Kanten durch die Entfernung der unerwarteten Box sowieso entfernt würden. Zusätzlich kann eine unerwartete Box immer das Ergebnis von ungenügenden Verbindungen sein.

5.3.2 Verfeinerung der Inkonsistenz-Typen

Auf Basis der in Abschnitt 2.6 beschriebenen Inkonsistenzen können verfeinerte Typen von Inkonsistenzen definiert werden. Dies dient dazu, zwei Ziele zu erreichen. Das erste Ziel ist, Gruppen von Inkonsistenzen, welche mit derselben Ursache zusammenhängen, in einem Typ zu vereinen. Dadurch wird die Anzahl der Inkonsistenzen reduziert und die einzelnen Inkonsistenzen werden aussagekräftiger. Das zweite Ziel ist, verschiedene Kombinationen von Inkonsistenz-Typen, welche dieselbe Ursache haben, einem einheitlichen Typen zuzuordnen. Es ist nicht das Ziel, die vorhandenen Inkonsistenz-Typen zu erstetzen oder einen verfeinerten Typen für alle möglichen Kombinationen von Inkonsistenzen zu erstellen. Daher sind die verfeinerten Typen ein Zusatz zu den vorhanden Typen.

Alle Typen sind von den realen Diagrammen motiviert. Dazu wurden die vom Ansatz erkannten Inkonsistenzen betrachtet. Zusätzlich wurden diese mit den im Goldstandard erwarteten Inkonsistenzen verglichen. Das bedeutet, dass diese verfeinerten Typen stark an die betrachteten Diagramme angepasst sind und in anderen realen Diagrammen nicht unbedingt vorkommen. Da der Goldstandard und die realen Diagramme auf diese Art verwendet worden sind, werden die verfeinerten Typen im Rahmen der Evaluation nicht betrachtet. Die verfeinerten Typen werden in der folgenden Liste aufgezählt.

1. `LineInversion`: Dieser Typ weist auf eine Linie hin, deren Richtung im Modell umgekehrt zu der im Diagramm ist. Dies wird erkannt, wenn eine Box eine `missing_line` zu einer anderen Box hat, welche eine `unexpected_line` zu der ersten Box hat.

2. `NameExtension`: Dieser Typ weist auf eine Box hin, deren erwarteter Name um weiteren Text erweitert wurde. Dies kann auf zwei Arten erkannt werden. Die erste ist, dass eine Box von einer `name_incosistency` betroffen ist, deren erwarteter Name ein Teil des tatsächlichen Namens ist. Die zweite ist, dass eine unerwartete Box (`unexpected_box`) einen Namen hat, welcher ein Teil des Namens eines fehlenden Modellelements (`missing_box`) ist.

3. `Casing`: Dieser Typ wird verwendet, um auf eine Inkonsistenz der verwendeten Konvention für Groß- und Kleinschreibung hinzuweisen. Alle Inkonsistenzen vom Typ `name_inconsistency`, deren erwarteter und tatsächlicher Name sich nur in der Groß- und Kleinschreibung unterscheiden, werden in einer einzigen Instanz dieses Typs zusammengefasst.

4. `Swap`: Dieser Typ weist auf das Vertauschen von zwei Boxen hin. Dies wird erkannt, wenn es zwei von einer `name_inconsistency` betroffene Boxen gibt, für die gilt: Der erwartete Name der ersten Box ist der tatsächliche Name der zweiten Box. Dabei wird nicht erwartet, dass der erwartete Name der zweiten Box der tatsächliche Name der ersten Box ist. Somit kann dieser Typ auch partielle Vertauschungen erkennen.

Gruppierung Das Ziel dieses Schrittes ist es, die gefundenen Inkonsistenzen zu gruppieren. Dabei wird nach der Box gruppiert, auf welche sich die Inkonsistenz bezieht. Bezieht sich die Inkonsistenz auf keine Box, so wird sie in keine Gruppe aufgenommen. Dies ist zum Beispiel bei `missing_box` und `Casing` der Fall. Es werden nur Gruppen mit mindestens zwei Inkonsistenzen erstellt.

5.3.3 Anwendung auf synthetischen Diagrammen

Die bereits in Abschnitt 4.2 beschriebenen Änderungen für synthetische Diagramme werden in diesem Abschnitt zur Untersuchung der Inkonsistenz-Erkennung verwendet. Für die Untersuchung wird die gesamte Pipeline verwendet. Dadurch kann die Auswirkung von unerwarteten Entscheidungen der Verbindungssuche auf die Inkonsistenz-Erkennung untersucht werden.

Bei synthetischen Architekturdiagrammen werden für alle Änderungsklassen bis auf `delete` alle Inkonsistenzen erkannt. Bei synthetischen Codediagrammen ist der F1-Score für erkannte Inkonsistenzen hoch, aber geringer als bei Architekturdiagrammen. Dies ist mit der bereits festgestellten geringeren Qualität der gefundenen Verbindungen in Zusammenhang mit den Codemodellen zu erklären. Die Änderung `delete` ist bei Codediagrammen ein Ausreißer, da hier ein Großteil der Inkonsistenzen nicht erkannt werden. Begründet ist dies darin, dass die Regel `AllModelEntitiesMustBeRepresented` für Codediagramme nicht verwendet wird. Sobald diese Regel aktiviert wird, werden ein Großteil der Inkonsistenzen bei `delete` erkannt. Für Codediagramme ist die Anwendung dieser Regel aber nicht sinnvoll, da alle betrachteten Codediagramme nur einen Ausschnitt des Modells darstellen. Des Weiteren sind die betrachteten Codemodelle mit mehreren hunderten Elementen sehr groß, sodass skizzierte Diagramme realistischerweise nur einen Bruchteil des Modells darstellen.

Werden alle Änderungsklassen gemischt und gleichzeitig angewendet, so sinkt die Qualität der Inkonsistenz-Erkennung bei Codediagrammen erheblich. Der Rückgang in der Qualität der Inkonsistenz-Erkennung ist stärker als der Rückgang in der Qualität der Verbindungssuche. Womöglich ist das Zusammenspiel von `create` und `delete` dafür verantwortlich. Die Modellelemente, welche von den durch `delete` entfernten Boxen dargestellt werden, können mit den durch `create` erstellten Boxen verbunden werden. Da nur ein Bruchteil der Boxen im Diagramm von synthetischen Änderungen betroffen ist und nur `create` und `delete` wiederum nur ein Teil dieser Änderungen sind, ist der dadurch erzeugte Effekt auf die Verbindungssuche gering. Gleichzeitig kann eine inkorrekte Verbindung zu mehreren inkorrekten Inkonsistenzen führen, da sowohl Name als auch alle ausgehenden Kanten dann womöglich als inkonsistent angesehen werden.

Für synthetische Codediagramme, welche das gesamte Codemodell darstellen, wird der Parameter `similarity_threshold` auf 0 gesetzt. Dies ist notwendig, um eine für die Inkonsistenz-Erkennung ausreichende Qualität der Verbindungssuche zu erreichen. Gleichzeitig hat dies eine Auswirkung auf die erkannten Inkonsistenzen bei Betrachtung von Codediagrammen, welche Ausschnitte des Modells darstellen. Wird der Parameter hier mit

dem Wert 0 verwendet, ist es für den Ansatz nicht möglich, dass eine Box nicht Teil einer Verbindung ist. Somit können neu hinzugefügte Boxen nicht als `unexpected_box` erkannt werden. Wird dagegen der Parameter mit einem Wert größer 0 verwendet, so werden die Boxen als `unexpected_box` erkannt, die allgemeine Qualität der Verbindungssuche sinkt aber. Für andere Änderungsklassen gibt es dann einen Rückgang der Qualität der Inkonsistenz-Erkennung. Die starke Bedeutung des Parameters bei synthetischen Diagrammen hängt potenziell damit zusammen, dass in synthetischen Diagrammen die Namen der sich entsprechenden Elemente in Modell und Diagramm identisch sind.

Insgesamt zeigt sich, dass die Qualität der Inkonsistenz-Erkennung stark von der Qualität der Verbindungssuche abhängt. Insbesondere inkorrekte Verbindungen führen im Vergleich zu fehlenden Verbindungen zu einer stärkeren Verschlechterung der Qualität. Ist die Qualität der Verbindungssuche hoch und die verwendeten Regeln decken die generierten Inkonsistenzen der Änderungen ab, dann ist die Qualität der Inkonsistenz-Erkennung hoch. Somit ist für eine Verbesserung der Inkonsistenz-Erkennung hauptsächlich die Verbesserung der Verbindungssuche und Auswahl der Regeln förderlich. Für die Auswahl der Regeln ist die Nutzung von synthetischen Inkonsistenzen nur begrenzt zielführend, da die verwendeten Änderungsklassen selbst impliziten Konsistenz-Regeln folgen. Somit ist keine Entdeckung von weiteren Regeln möglich, welche in der Realität vorkommende Spezialfälle abdecken. Mit Spezialfällen ist gemeint, dass reale Inkonsistenzen möglicherweise Mustern folgen, welche durch die zufälligen synthetischen Änderungen nicht reproduziert werden.

5.3.4 Zwischenevaluation mit realen Diagrammen

Tabelle 5.9 zeigt die Ergebnisse der Zwischenevaluation der Inkonsistenz-Erkennung. Dabei wurden nur die Inkonsistenzen für den im Goldstandard erwarteten Typ betrachtet.

Zu erkennen ist, dass der F1-Score für `TS` mit 0,52 besonders gering und für `BBB` mit 1,00 besonders hoch ist. Der F1-Score der Verbindungssuche scheint von Bedeutung zu sein, aber auch andere Eigenschaften unabhängig von den Metriken ist wichtig. Dies ist daran zu erkennen, dass die Verbindungssuche bei `BBB` und `TM_A` die gleiche Qualität erreicht, dies aber für die Inkonsistenz-Erkennung nicht der Fall ist.

Bei Diagramm `TM_P` wählt die Typ-Erkennung beide Diagramm-Typen aus, somit wird die Verbindungssuche und Inkonsistenz-Erkennung mit beiden Modellen durchgeführt. Eine gemeinsame Betrachtung aller gefundenen Inkonsistenzen mit beiden Modellen ist in diesem Fall nicht sinnvoll und auch im Allgemeinen möglicherweise nicht hilfreich. Der Grund dafür ist, dass die gefundenen Inkonsistenzen widersprüchlich sein können, wenn Codemodell und Architekturmodell selbst nicht konsistent sind. Es ist daher in solchen Fällen bei Verwendung des Ansatzes eine manuelle Entscheidung nötig, welches Modell verwendet werden soll. Gleichzeitig kann eine gemeinsame Betrachtung der Inkonsistenzen mehrerer Modelle sinnvoll sein, wenn ein Diagramm eine Mischung aus Code- und Architekturdiagramm darstellt. Die Betrachtung von Diagrammen, welche mehrere Modelle abbilden, geht über diesen Ansatz hinaus. Um solche Mischformen mit

Diagramm	Präzision	Ausbeute	F1
BBB	1,00	1,00	1,00
TM_A	0,77	0,77	0,77
TM_P	0,65	0,65	0,65
TM_U	0,63	0,86	0,73
TS	0,67	0,42	0,52
MS	0,79	0,73	0,76

Tabelle 5.9: Evaluation der Inkonsistenz-Erkennung; `similarity_threshold_architecture`: 0,6; `similarity_threshold_code`: 0,8; `match_threshold`: 0,05; `match_difference`: 0,05; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,68; `similarity_threshold`: 0,12; git-tag: `eval-stage3-v6`

dem vorgestellten Ansatz zu betrachteten, ist ein gemischtes Modell nötig. Ein gemischtes Modell würde sowohl Code- als auch Architektur-Elemente enthalten. Zur Erstellung eines gemischten Modells aus Code- und Architekturmodell könnte ein Ansatz wie der von Telge [27] als Ausgangspunkt dienen. Darin werden Nachverfolgbarkeitsverbindungen zwischen Code- und Architekturmodell automatisiert ermittelt.

Anzumerken ist, dass auch laut Goldstandard unerwartete Inkonsistenzen hilfreich bei der Beseitigung von tatsächlichen Inkonsistenzen zwischen Diagramm und Modell sein können. So kann zum Beispiel ein inkonsistenter Name, im Goldstandard eine `name_inconsistency`, durch großen Unterschied und inkonsistente Nachbarschaft dazu führen, dass keine Verbindung gefunden wird. Dann ist es dennoch möglich, dass der Ansatz mit einer Kombination von `missing_box` und `unexpected_box` auf die Inkonsistenz hinweist. Ähnliche Szenarien werden in Unterabschnitt 5.3.2 behandelt. Es werden dort erweiterte Inkonsistenz-Typen dargestellt, welche auf Kombinationen der einfachen Typen basieren. Dadurch müssen Nutzer weniger verschiedene Inkonsistenzen betrachten. Zusätzlich werden einige verschiedene Kombinationen mit gleicher Bedeutung zu einem Typ zusammengefasst.

Eine Idee zur Verbesserung der Nutzbarkeit wäre, für jede Inkonsistenz einen Konfidenz-Wert zu bestimmen. Diesen Wert könnte ein Nutzer dann bei der Auswertung der Inkonsistenzen berücksichtigen. Zur Bestimmung eines solchen Wertes könnten die von *graph matching* berechneten Ähnlichkeiten verwendet werden. Gegen die Verwendung dieser Werte spricht, dass die berechneten Ähnlichkeitswerte keinem intuitiven Maß entsprechen. Die Ähnlichkeiten erwarteter Verbindungen füllen einen großen Teil des Wertebereichs $[0; 1]$ aus. Für keines der betrachteten Diagramme konnte eine Häufung von Werten in einem Teilbereich des Wertebereichs festgestellt werden. Unerwartete Verbindungen fallen bei den betrachteten Diagrammen in einen niedrigen, aber nicht vom Bereich der erwarteten Verbindungen klar abgrenzbaren Bereich. Zusätzlich können Diagramme mit vielen Inkonsistenzen zu insgesamt niedrigeren Ähnlichkeiten führen. Daher wird diese Idee nicht weiter verfolgt.

5.3.5 Zwischenevaluation mit erweitertem Codemodell

Bei der Erstellung der Goldstandards ist aufgefallen, dass Linien in den betrachteten Codediagrammen mit anderer Bedeutung verwendet werden, als vom Ansatz berücksichtigt. Der Ansatz erwartet Linien zwischen Klassen und Schnittstellen, um Vererbungs- und Implementierungsbeziehungen darzustellen. Alle anderen Linien, insbesondere Linien zwischen Paketen, werden als Inkonsistenz betrachtet. In den betrachteten Diagrammen werden Linien dagegen zur Darstellung von Abhängigkeiten zwischen Klassen, Schnittstellen und Paketen verwendet.

Diagramm	`...inconsistency`		`missing...`		`unexpected...`	
	`name`	`hierarchy`	`line`	`box`	`line`	`box`
`BBB`	11	0	3	0	7	1
`TM_A`	8	0	3	0	0	7
`TM_P`	21	0	0	5	14	3
`TM_U`	2	0	0	0	5	15
`TS`	0	0	4	5	10	0
`MS`	2	0	8	4	0	1

Tabelle 5.10: Anzahl Inkonsistenzen je Typ im Goldstandard

Davon betroffen sind die Diagramme `TM_P` und `TM_U`. Tabelle 5.10 zeigt die Anzahl der Inkonsistenzen je Typ im Goldstandard. Alle in den Codediagrammen vorkommenden Linien werden im Goldstandard als `unexpected_line` erfasst, da diese keine Vererbung- oder Implementierungsbeziehung darstellen. Um die Abhängigkeiten in den Diagrammen zu berücksichtigen, wird das Codemodell entsprechend der Beschreibung in Abschnitt 2.5 erweitert.

Diagramm	**Präzision**	**Ausbeute**	**F1**
`TM_P`	0,65	0,53	0,59
`TM_U`	0,68	0,85	0,76

Tabelle 5.11: Evaluation der Inkonsistenz-Erkennung mit erweitertem Codemodell; `similarity_threshold_architecture`: 0,6; `similarity_threshold_code`: 0,8; `match_threshold`: 0,05; `match_difference`: 0,05; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,68; `similarity_threshold`: 0,12; git-tag: `eval-stage3-v7`

Die Auswirkungen auf den gesamten Ansatz und die Ergebnisse der Inkonsistenz-Erkennung sind in Tabelle 5.11 dargestellt. Da sich das zugrundelegende Modell geändert hat, wird der Goldstandard entsprechend angepasst. Im Fall von Diagramm `TM_P` werden dabei etwa 40 mehr Inkonsistenzen vom Typen `missing_line` erwartet. Für `TM_P` ist die Qualität der Inkonsistenz-Erkennung gesunken, für `TM_U` ist die Qualität gestiegen. Die Erweiterung des Codemodells dient nicht der Verbesserung der gemessenen Qualität der Inkonsistenz-Erkennung, sondern der Verbesserung der Nützlichkeit der gefundenen Inkonsistenzen. Die Nützlichkeit wird in dieser Arbeit nicht gemessen.

6 Evaluation

In diesem Abschnitt werden die drei Schritte des Ansatzes mit realen Diagrammen evaluiert. Die verwendeten realen Diagramme sind in Abschnitt 4.1 aufgelistet.

Inhalt der Goldstandards Für alle drei Schritte und jedes reale Diagramm benötigt es Goldstandards. Für Schritt 1 gibt der Goldstandard für jedes Element in einem Diagramm die möglicherweise leeren Mengen der Zuordnungen zu Modellen (Code, Architektur) und der Element-Arten in den Modellen an. Die Rollen sind dabei `architecture:component`, `architecture:interface`, `code:package`, `code:class` und `code:interface`. Für Schritt 2 gibt der Goldstandard für ein Diagramm eine Menge aller Verbindungen an, wobei eine Verbindung ein Paar aus einem Knoten im Diagramm und einer Entität im relevanten Modell ist. Für Schritt 3 besteht der Goldstandard aus einer Liste aller zu findenden Inkonsistenzen. Die Inkonsistenz-Typen sind wie in Abschnitt 2.6 beschrieben die folgenden: `name_inconsistency`, `hierarchy_inconsistency`, `missing_line`, `unexpected_line`, `missing_box` und `unexpected_box`. Dabei beinhaltet eine Inkonsistenz die betroffenen Graph-Bestandteile, Modell-Bestandteile und den Inkonsistenz-Typen, sowie je nach Typ mögliche Zusatz-Informationen.

Erstellung der Goldstandards Die realen Diagramme beinhalten bereits Inkonsistenzen. Daher ist die Erstellung der Goldstandards nicht eindeutig. Es wurden alle Elemente als Boxen interpretiert, selbst wenn diese keine Box, sondern zum Beispiel das Symbol für einen Akteur sind. Im folgenden werden einige Entscheidungen bei der Erstellung der Goldstandards beschrieben. Die in Unterabschnitt 5.3.1 beschriebenen Regeln sind an die hier getroffenen Entscheidungen angelehnt.

- In einem der Diagramme ist ein Element des Modells im Diagramm mit zwei Boxen dargestellt. Hier wurde nur eines der Elemente als Teil einer Verbindung interpretiert. Dieses Diagramm ist somit inkonsistent, auch wenn die Zweiteilung des Elements im Diagramm beabsichtigt sein könnte.
- Das Architekturmodell beinhaltet Komponenten und Schnittstellen. Für jede Komponente ist bekannt, welche Schnittstellen sie anbietet und welche sie nutzt. Es können mehrere Komponenten die gleiche Schnittstelle anbieten und das Modell beinhaltet keine Information darüber, mit welcher Komponente eine Abhängigkeit aufgelöst wird. Daher wird eine Kante zu jeder bereitstellenden Komponente erwartet, ein Fehlen im Diagramm ist eine Inkonsistenz. Ausgenommen hiervon sind Kanten von einer Komponente zu sich selbst, diese werden nicht erwartet.

- Für Architekturmodelle wird erwartetet, dass jede Komponente im Diagramm dargestellt ist, ein Fehlen ist eine Inkonsistenz. Fehlt ein Element im Diagramm, so sind alle erwarteten Kanten zu diesem Element keine Inkonsistenz, da diese ohne das Element nicht dargestellt werden können.
- Für Codemodelle wird erwartet, dass sobald ein Unterpaket eines Paketes im Diagramm dargestellt ist und das Unterpaket keine Boxen beinhaltet, auch alle anderen Unterpakete des Paketes dargestellt sind.
- Ist eine Box im Diagramm unerwartet, so werden keine Kanten zu dieser Box als Inkonsistenz gewertet. Durch Entfernen der Box würden diese Kannten ebenfalls entfernt werden, somit ist keine explizite Kennzeichnung notwendig.

Neben den einzelnen Evaluationen jedes Schrittes wird auch der gesamte Ansatz am Stück untersucht. Statt des entsprechenden Goldstandards werden Schritt 2 und 3 dann mit der Ausgabe des vorherigen Schrittes durchgeführt.

Validität Für die Evaluation werden sechs verschiedene Diagramme verwendet, von denen nur zwei Codemodelle darstellen. Somit ist nur ein geringer Satz an Daten für die Evaluation vorhanden. Synthetische Diagramme wurden während der Entwicklung eingesetzt, doch diese können reale Diagramme nicht ersetzen. Des Weiteren ist der Goldstandard vom Autor dieser Arbeit erstellt worden. Da die Interpretation von Inkonsistenzen abhängig vom Betrachter sein kann, können andere Personen zu anderen Ergebnissen kommen.

Forschungsfragen Für jeden Schritt werden Präzision, Ausbeute, und F1-Score untersucht. In Schritt 1 wird zusätzlich die vom Ansatz ermittelte Übereinstimmung von Diagramm und dem darin dargestellten Modell verglichen mit der Übereinstimmung mit dem anderen Modell. Da der *graph matching*-Ansatz in Schritt 2 die Struktur berücksichtigt, sollten einzelne Umbenennungen die Verbindungssuche nicht beeinträchtigen. Hier ist zu untersuchen, wie stabil der Ansatz bei Bezeichner-Inkonsistenz ist. Dies gibt darüber Aufschluss, wie gut die Struktur des Diagramms und der Modelle ausgenutzt wird. Damit verwandt ist die Untersuchung der Abhängigkeit von der Kantenstruktur und die Stabilität bezüglich Strukturänderung. Dazu wird der Ansatz auf den gleichen Eingaben, aber mit zunehmend fehlenden Kanten untersucht werden.

G Automatisiertes Bestimmen des Diagramm-Typen (Schritt 1)

  Q Wie gut kann der Typ bestimmt werden?

    M Präzision, Ausbeute, F1-Score der gefundenen Vorkommen

    M Differenz der Übereinstimmungen

G Finden von Verbindungen zwischen Diagramm- und Modellelementen (Schritt 2)

  Q Wie gut können Verbindungen gefunden werden?

    M Präzision, Ausbeute, F1-Score der gefundenen Verbindungen

- Q Wie stabil ist die Verbindungssuche bei Bezeichner-Inkonsistenz?
  - M Vergleich der vorherigen Metriken bei zunehmender Umbenennung im Diagramm
- Q Wie stabil ist die Verbindungssuche bei Struktur-Inkonsistenz?
  - M Vergleich der vorherigen Metriken bei zunehmender struktureller Veränderung im Diagramm

G Erkennen von Inkonsistenzen zwischen Diagramm und Modell (Schritt 3)

- Q Wie gut werden Struktur-Inkonsistenzen auf Basis von Verbindungen aus dem Goldstandard?
  - M Präzision, Ausbeute, F1-Score der erkannten Verbindungen
- Q Wie gut werden Bezeichner-Inkonsistenzen auf Basis von Verbindungen aus dem Goldstandard erkannt?
  - M Präzision, Ausbeute, F1-Score der erkannten Verbindungen
- Q Wie gut werden Inkonsistenzen mit dem Gesamt-Ansatz erkannt?
  - M Präzision, Ausbeute, F1-Score der erkannten Verbindungen

6.1 Typ-Erkennung

Um die Typ-Erkennung zu evaluieren, wird jedes Diagramm sowie die beiden Modelle an den Ansatz übergeben. Nur die Ergebnisse des ersten Schrittes werden betrachtet. Wie in Unterabschnitt 5.1.1 beschrieben, sucht der Ansatz für jede Box im Diagramm nach entsprechenden Elementen in den Modellen. Diese Elemente werden als Vorkommen der Box im Modell bezeichnet. Der Goldstandard definiert eine Menge erwarteter Vorkommen, diese werden mit den gefundenen Vorkommen verglichen, um Präzision, Ausbeute und F1-Score zu bestimmen. Zusätzlich zu den Metriken, welche sich auf die einzelnen Vorkommen beziehen, muss auch die tatsächliche Klassifikation betrachtet werden. Auch diese wird mit dem Goldstandard verglichen.

Im Rahmen der Zwischenevaluationen wurden zwei potenzielle Textähnlichkeitsfunktionen identifiziert. Diese sind die Levenshtein-Ähnlichkeit und die in Unterunterabschnitt 5.1.5.2 beschriebene angepasste Jaccard-Ähnlichkeit. Beide werden im Folgenden evaluiert. Um nicht durch die Wahl der Parameter eine der beiden Funktionen zu bevorzugen, werden erst für beide Funktionen die jeweils günstigeren Parameter ermittelt.

6.1.1 Wahl der Parameter

Im Folgenden werden die zwei Parameter `similarity_threshold_architecture` und `similarity_threshold_code` untersucht. Diese Parameter bestimmen einen Schwellenwert, über dem die Ähnlichkeit zweier Elemente beziehungsweise ihrer Bezeichner liegen muss, damit das Diagramm-Element als Vorkommen des Modellelements gewertet wird. Die Untersuchung der Parameter wird sowohl mit der Levenshtein-Ähnlichkeit als auch mit der angepassten Jaccard-Funktion durchgeführt.

Für beide Funktionen sind die Ergebnisse in je zwei Abbildungen dargestellt. Jede Abbildung betrachtet die drei Metriken `F1`, `match_gs` und `match_other` abhängig vom Wert des Parameters. Die Metrik `F1` ist der F1-Score der gefundenen Vorkommen bezüglich des Goldstandards. Die Metrik `match_gs` ist die ermittelte Übereinstimmung des Diagramms mit dem im Goldstandard zugeordneten Modell. Die Metrik `match_other` ist die ermittelte Übereinstimmung des Diagramms mit dem entsprechend anderen Modell. Ist im Goldstandard das Codemodell vorgesehen, so ist das Architekturmodell das andere Modell, ansonsten umgekehrt.

Abbildung 6.1 und Abbildung 6.2 zeigen die Ergebnisse für die Levenshtein-Distanz. Gemäß der Untersuchung ist für `similarity_threshold_architecture` ein Wert von 0,5 und für `similarity_threshold_code` ein Wert von 0,8 geeignet, um den F1-Score zu maximieren.

Abbildung 6.3 und Abbildung 6.4 zeigen die Ergebnisse für die angepasste Jaccard-Funktion. Gemäß der Untersuchung ist für `similarity_threshold_architecture` ein Wert von 0,6 und für `similarity_threshold_code` ein Wert von 0,8 geeignet, um den F1-Score zu maximieren.

6.1.2 Vergleich der Ähnlichkeitsfunktionen

In diesem Abschnitt werden die beiden Ähnlichkeitsfunktionen Levenshtein-Distanz und angepasste Jaccard-Funktion verglichen. Dabei werden für jede Funktion die entsprechend in Unterabschnitt 6.1.1 ermittelten Parameter verwendet.

Tabelle 6.1 zeigt die Ergebnisse für die angepasste Levenshtein-Distanz. Für Diagramm `TM_P` wird statt dem im Goldstandard zugeordneten Codemodell das Architekturmodell gewählt. Für Diagramm `TS` werden beide Modelle gewählt.

Tabelle 6.2 zeigt die Ergebnisse für die angepasste Jaccard-Funktion. Für jedes Diagramm bis auf `TM_P` ist das im Goldstandard zugeordnete Modell das Modell mit der höheren Übereinstimmung. Bei `TM_P` ist die Übereinstimmung für beide Modelle gleich. Wegen der geringen Differenz der Übereinstimmungen wählt der Ansatz in diesem Fall beide Modelle.

Der durchschnittliche F1-Score mit der Levenshtein-Distanz ist etwa 0,76. Mit der angepassten Jaccard-Funktion ist der durchschnittliche F1-Score ungefähr 0,73. Die angepasste Jaccard-Funktion ist somit minimal schlechter geeignet, um die Vorkommen in den Modellen zu finden. Gleichzeitig wird mit der Levenshtein-Funktion eine inkorrekte Klassifikation

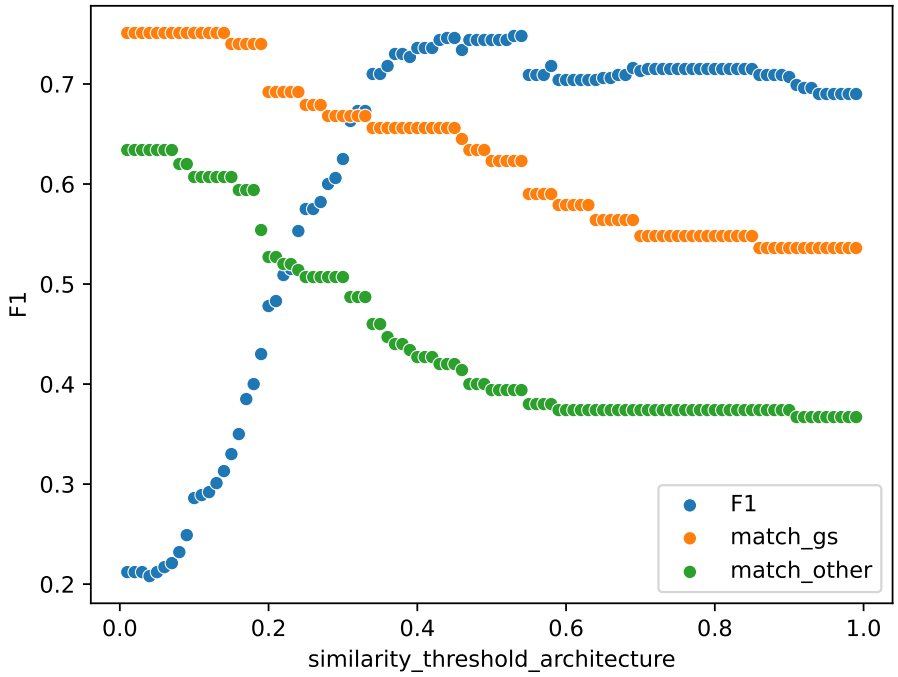

Abbildung 6.1: Auswahl des Parameters `similarity_threshold_architecture` für Typ-Erkennung mit Levenshtein-Ähnlichkeit; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v13`

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,92	0,77	0,85	0,81	0,83
TM_A	0,27	0,27	1,00	0,26	0,41
TM_P	0,40	-0,12	0,84	0,58	0,69
TM_U	0,36	0,28	1,00	0,93	0,96
TS	1,00	**0**	0,81	1,00	0,89
MS	0,91	0,18	0,82	0,75	0,78

Tabelle 6.1: Evaluation der Typ-Erkennung mit angepasster Levenshtein-Distanz; `similarity_threshold_architecture`: 0,5; `similarity_threshold_code`: 0,8; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v13`

durchgeführt, also nicht der Modell-Typ gewählt, welcher im Goldstandard vorgesehen ist. Da korrekte Klassifikationen für die folgenden Schritte der Pipeline wichtig sind, wird die angepasste Jaccard-Funktion für das weitere Vorgehen verwendet.

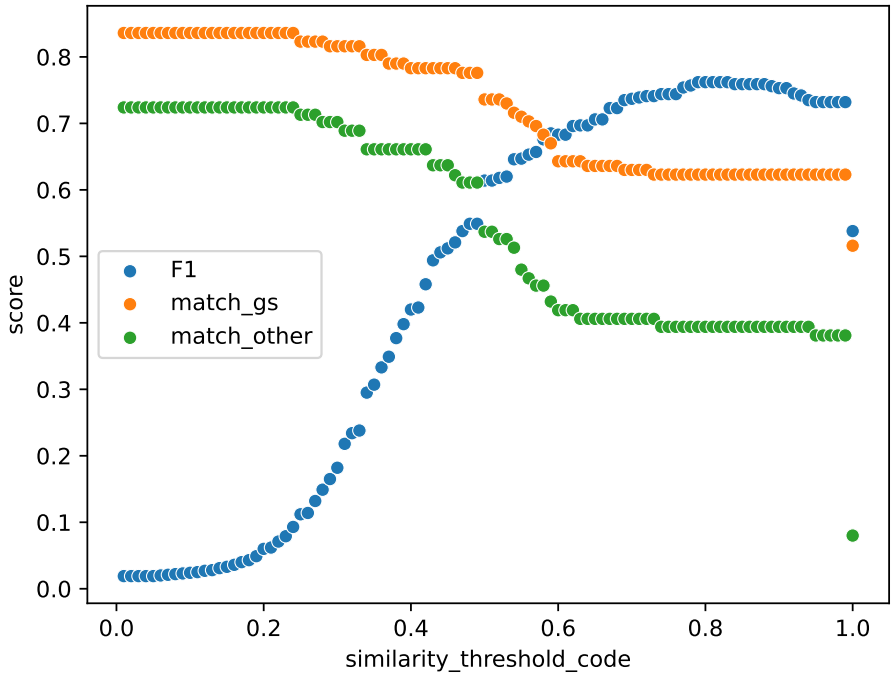

Abbildung 6.2: Auswahl des Parameters `similarity_threshold_code` für Typ-Erkennung mit Levenshtein-Ähnlichkeit; `similarity_threshold_architecture`: 0,5; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v13`

Diagramm	Übereinstimmung	Δ	Präzision	Ausbeute	F1
BBB	0,69	0,38	0,89	0,72	0,80
TM_A	0,33	0,33	1,00	0,35	0,52
TM_P	0,52	**0**	0,63	0,72	0,68
TM_U	0,48	0,40	0,67	1,00	0,80
TS	1,00	0,17	0,80	0,94	0,86
MS	0,82	0,09	0,68	0,72	0,70

Tabelle 6.2: Evaluation der Typ-Erkennung mit angepasster Jaccard-Funktion; `similarity_threshold_architecture`: 0,6; `similarity_threshold_code`: 0,8; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v18`

6.2 Verbindungssuche

Um die Verbindungssuche zu evaluieren, werden jedes Diagramm sowie die beiden Modelle dem Ansatz übergeben. Die Typ-Erkennung wird übersprungen und der erwartete Typ direkt aus dem Goldstandard übernommen. Die gefundenen Verbindungen werden mit denen aus dem Goldstandard verglichen.

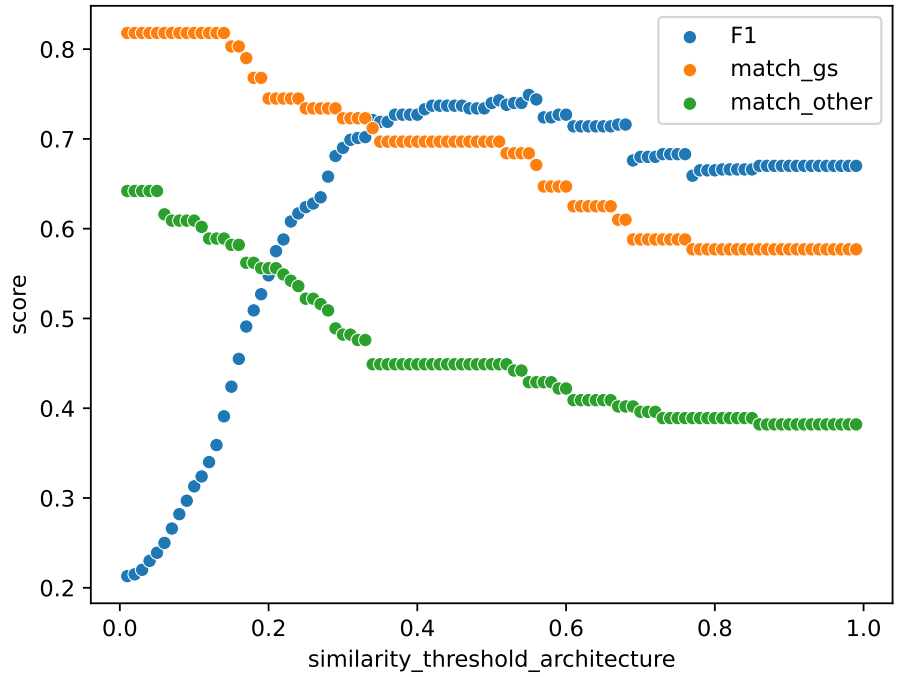

Abbildung 6.3: Auswahl des Parameters `similarity_threshold_architecture` für Typ-Erkennung mit angepasster Jaccard-Ähnlichkeit; `similarity_threshold_code`: 0,75; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v18`

6.2.1 Wahl der Parameter

Zur Wahl der Parameter werden für jeden Parameter einzeln jedes Diagramm mit dem Ansatz ausgewertet. Es werden der minimale und maximale F1-Score aller Diagramme in einer Ausführung sowie der Durchschnitt und der gewichtete Durchschnitt über eine Ausführung gezeigt. Der gewichtete Durchschnitt nutzt die Anzahl erwarteter Verbindungen als Gewicht. In der Evaluation verschiedener Werte für `epsilon`, dargestellt in Abbildung 6.5, ist ein Plateau zu erkennen. Der derzeit gewählte Wert von 1,0 liegt in diesem Plateau und wird beibehalten. In der Evaluation des `text_similarity_threshold`, siehe Abbildung 6.6, ist das Minimum und der Durchschnitt bei 0,58 maximal. Daher wird der Parameter auf 0,58 gesetzt. In der Evaluation des `similarity_threshold`, in Abbildung 6.7, ist das Minimum und der Durchschnitt bei 0,12 maximal, daher wird dieser Wert verwendet.

Mit den neuen Parametern ergibt die Auswertung der Diagramme die Ergebnisse in Tabelle 6.3. Diese wird mit der Zwischenevaluation in Unterabschnitt 5.2.6 verglichen. Für `TM_A` und `TM_P` ist die Qualität gesunken. Für `TM_U` ist die Qualität in größerem Maße gestiegen. Dies zeigt, dass die Parameter nicht für alle Diagramme passend sind. Wie in den Untersuchungen bereits erkannt, hängen die Auswirkungen der Parameter von den

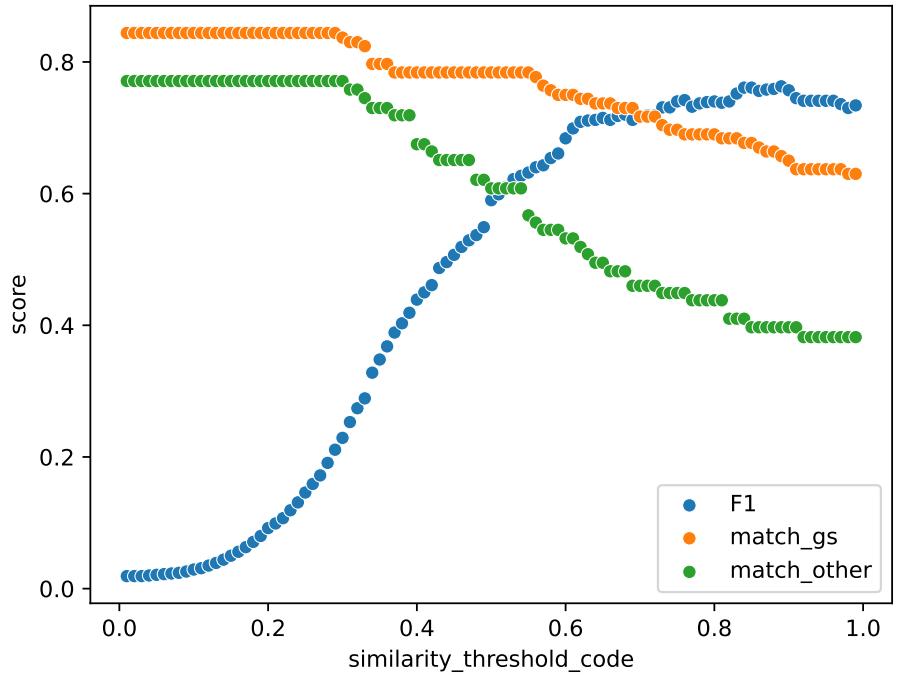

Abbildung 6.4: Auswahl des Parameters `similarity_threshold_code` für Typ-Erkennung mit angepasster Jaccard-Ähnlichkeit; `similarity_threshold_architecture`: 0,5; `match_threshold`: 0,05; `match_difference`: 0,05; git-tag: `eval-stage1-v18`

Änderungsklassen ab. Die verschiedenen Diagramme haben unterschiedliche Zusammensetzungen der Änderungsklassen. Der Grund dafür ist, dass dies reale Diagramme sind, welche als Teil unterschiedlicher Entwicklungsprozesse entstanden sind.

6.2.2 Stabilität

Im Folgenden wird die Stabilität der Verbindungssuche evaluiert. In der Numerik ist ein Algorithmus stabil, wenn kleine Änderungen beziehungsweise Fehler der Eingabe nur kleine Änderungen der Ausgabe zur Folge haben [15]. Dabei ist die Interpretation von ‚klein' abhängig von der Eingabe, Ausgabe und der Anwendung. Für diese Evaluation bedeutet Stabilität, dass die Qualität (F1-Score) der Verbindungssuchen (*matching*) bei inkonsistenten Eingaben schwächer abnimmt, als die Inkonsistenz der Eingabe zunimmt. Um die Stabilität zu untersuchen, wird eine Auswahl von den beschriebenen Änderungen, siehe Abschnitt 2.6, auf die Diagramme angewendet. Dabei werden die erwarteten Verbindungen entsprechend angepasst und eine Liste der Inkonsistenzen erstellt.

Für die folgende Evaluation werden zwei verschiedene Änderungsklassen getrennt betrachtet. Die erste Art verursacht Bezeichner-Inkonsistenz, indem der Name zufälliger

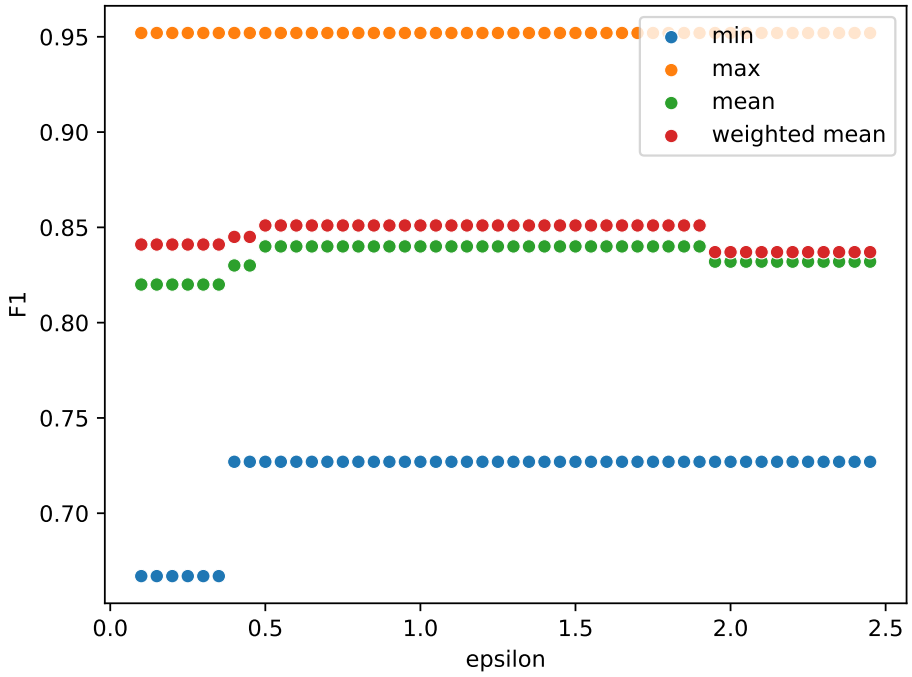

Abbildung 6.5: Auswahl des Parameters `epsilon` für Verbindungssuche; `max_iterations`: 100; `text_similarity_threshold`: 0,5; `similarity_threshold`: 0,1; git-tag: `eval-stage2-v22`

Boxen zu einer zufälligen Zeichenfolge geändert wird. Die zweite Art verursacht Struktur-Inkonsistenz, indem ausgehende Linien von zufällig ausgewählten Boxen entfernt werden. In beiden Fällen werden für verschiedene Änderungsverhältnisse von 0 bis 1 angepasste Diagramme generiert. Das Verhältnis gibt an, welcher Teil der Boxen im Diagramm von einer Änderung betroffen ist. Für jedes Verhältnis und Diagramm werden zehn Varianten erstellt.

Die Ergebnisse für die Bezeichner-Änderung sind in Abbildung 6.8 dargestellt. Die durchschnittliche Qualität sinkt mit zunehmendem Anteil der geänderten Bezeichner. Bei höherem Veränderung-Verhältnis sinkt die Qualität stärker. Ab einem Anteil von 0,3 gibt es Diagramme in denen die Qualität auf 0,0 sinkt. Gleichzeitig ist zu erkennen, dass es auch bei hohem Anteil noch Diagramme gibt, in denen die Qualität über 0,8 liegt. Dies kann dadurch erklärt werden, dass die bereits vorhandenen, unterschiedlichen Inkonsistenzen in den Diagrammen die Qualität entsprechend unterschiedlich beeinflussen. Insgesamt ist zu erkennen, dass die Stabilität gegenüber Umbenennungen von dem betrachteten Änderungs-Verhältnis abhängt. Bei einem niedrigen Verhältnis ist der Rückgang in der Qualität relativ kleiner, somit stabil.

Die Auswirkung von entfernten Linien ist in Abbildung 6.9 dargestellt. Es ist zu erkennen, dass die Auswirkung auf die durchschnittliche Qualität geringer ausfällt. Der Ansatz ist

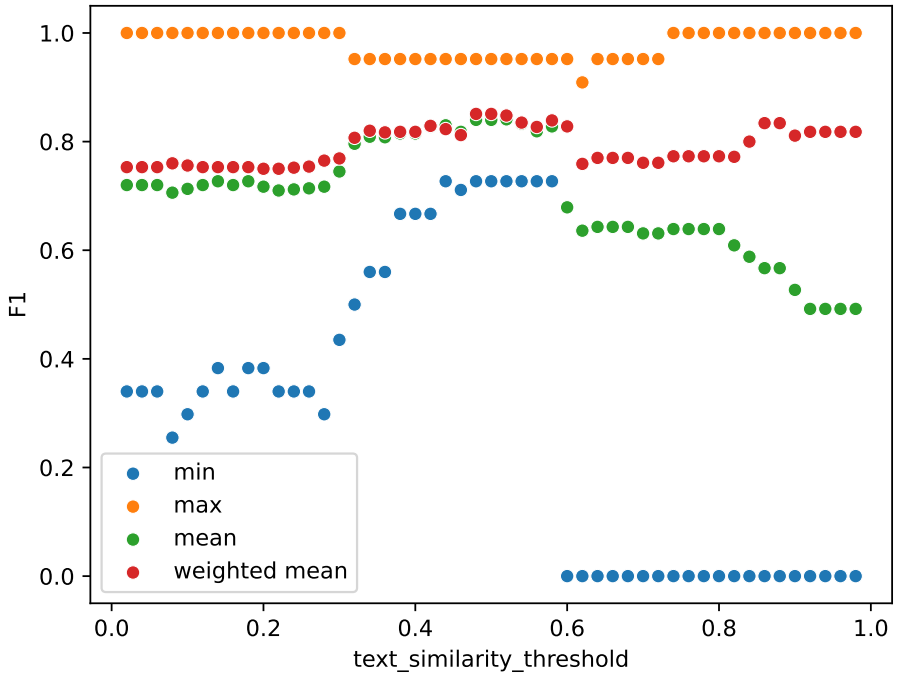

Abbildung 6.6: Auswahl des Parameters `text_similarity_threshold` für Verbindungssuche;
`epsilon`: 1,0; `max_iterations`: 100; `similarity_threshold`: 0,1; git-tag: `eval-stage2-v22`

selbst bei sehr hohen Verhältnissen stabil gegenüber den durch entfernte Linien eingebrachten Fehler. Dies könnte darauf hinweisen, dass der Ansatz die Bezeichner besser nutzen kann als die Struktur. Dadurch fällt ein Rückgang in der Aussagekraft der Bezeichner stärker ins Gewicht. Es stellt sich die Frage, ob das Nutzen der Struktur mit dem beschriebenen Ansatz förderlich ist. Möglicherweise wird die Struktur nicht ausreichend oder effektiv genug genutzt. Es ist aber auch möglich, dass reale Diagramme grundlegend eher strukturelle als namentliche Unterschiede zu den dargestellten Modellen haben.

6.2.3 Vorteil von Graph Matching

Um zu evaluieren, ob die Anwendung des *graph matching*s einen Vorteil liefert, wird der Ansatz ohne dieses evaluiert. Dazu wird der Parameter `max_iterations` auf 0 gesetzt. Die Ergebnisse sind in Tabelle 6.4 dargestellt. Die Präzision ist in den meisten Diagrammen gestiegen, während die Ausbeute meist gesunken ist. Für `BBB`, `TM_A`, `TM_P` und `TM_U` ist die Qualität gesunken. Für `TS` und `MS` ist die Qualität gestiegen. Der Ansatz kann also durch die Anwendung von *graph matching* einen Vorteil erzielen. Dies ist aber nicht für alle

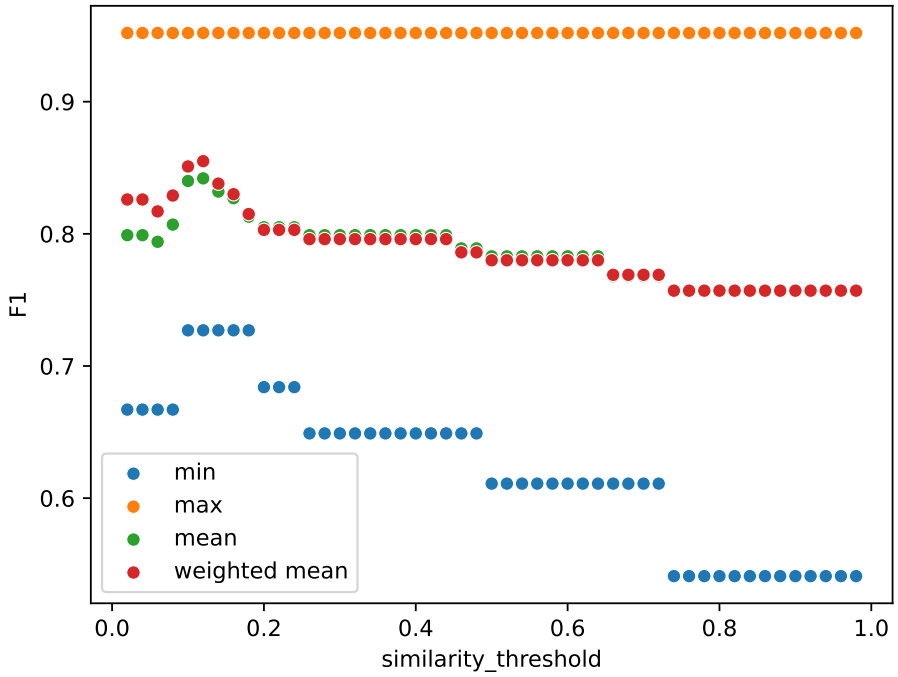

Abbildung 6.7: Auswahl des Parameter `similarity_threshold` für Verbindungssuche; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,5; git-tag: `eval-stage2-v22`

Diagramme der Fall, auch ein Nachteil ist möglich. Insbesondere in Diagrammen, in denen die Bezeichner mit denen im Modell übereinstimmen, ist ein Nachteil möglich.

Um die Qualität bei Diagrammen zu erhöhen, bei denen der Ansatz mit *graph matching* einen Nachteil hat, könnten Bezeichner stärker berücksichtigt werden. Der Nachteil entsteht dadurch, dass die Bezeichner größtenteils direkte Entsprechung im Modell haben, während die Struktur weniger übereinstimmend ist. Werden Verbindungen auf Grundlage der Bezeichner erstellt, bevor der *graph matching*-Algorithmus angewendet wird, geht diese Information nicht mehr verloren. Gleichzeitig kann die Struktur aber nicht mehr effektiv ausgenutzt werden. Eine priorisierte Einbeziehung der Bezeichner könnte verhindern, auf Basis der Struktur Elemente zu verbinden, deren Bezeichner zum Beispiel vertauscht wurden. Zusätzlich ist es möglich, selbst bei niedriger Qualität der Verbindungen, sinnvoll Inkonsistenzen zu erkennen. Sind zum Beispiel zwei Bezeichner vertauscht, so sind zwei Interpretationen möglich. Wird auf Basis der Bezeichner verbunden, so ist die Inkonsistenz, dass die Struktur, also die Nachbarschaft der Elemente, nicht übereinstimmt. Wird auf Basis der Struktur verbunden, so ist die Inkonsistenz, dass die Bezeichner nicht übereinstimmen. Beides kann einen Hinweis auf die tatsächliche Inkonsistenz geben.

Diagramm	Präzision	Ausbeute	F1
BBB	1,00	0,75	0,86
TM_A	1,00	0,63	0,77
TM_P	0,81	0,77	0,79
TM_U	0,77	1,00	0,87
TS	0,80	0,67	0,73
MS	0,91	1,00	0,95

Tabelle 6.3: Evaluation der Verbindungssuche mit neuen Parametern; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,58; `similarity_threshold`: 0,12; git-tag: `eval-stage2-v22`

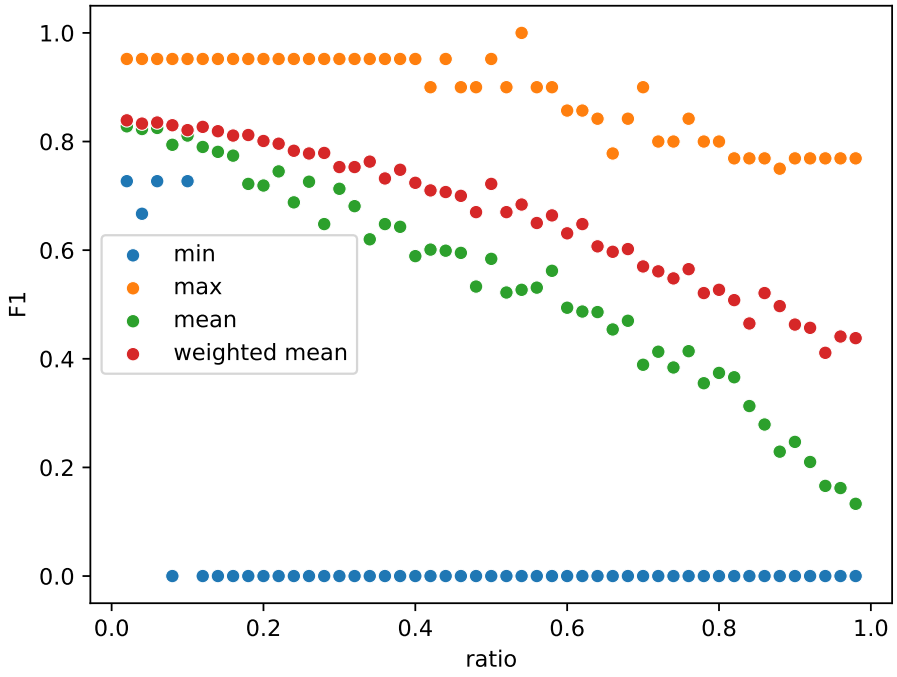

Abbildung 6.8: Evaluation der Verbindungssuche bei zunehmender Umbenennung; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,58; `similarity_threshold`: 0,12; git-tag: `eval-stage2-v22`

6.2.4 Verwendung der angepassten Jaccard-Ähnlichkeit

Genauso wie in Unterabschnitt 6.1.2 für die Typ-Erkennung die Levenshtein-Distanz mit der angepassten Jaccard-Funktion verglichen und ersetzt wurde, ist dies auch für die Verbindungssuche möglich.

Der Parameter `text_similarity_threshold` wird weiter verwendet, bedarf wegen der veränderten Verteilung der Ähnlichkeiten aber einer Anpassung. In Abbildung 6.10 sind

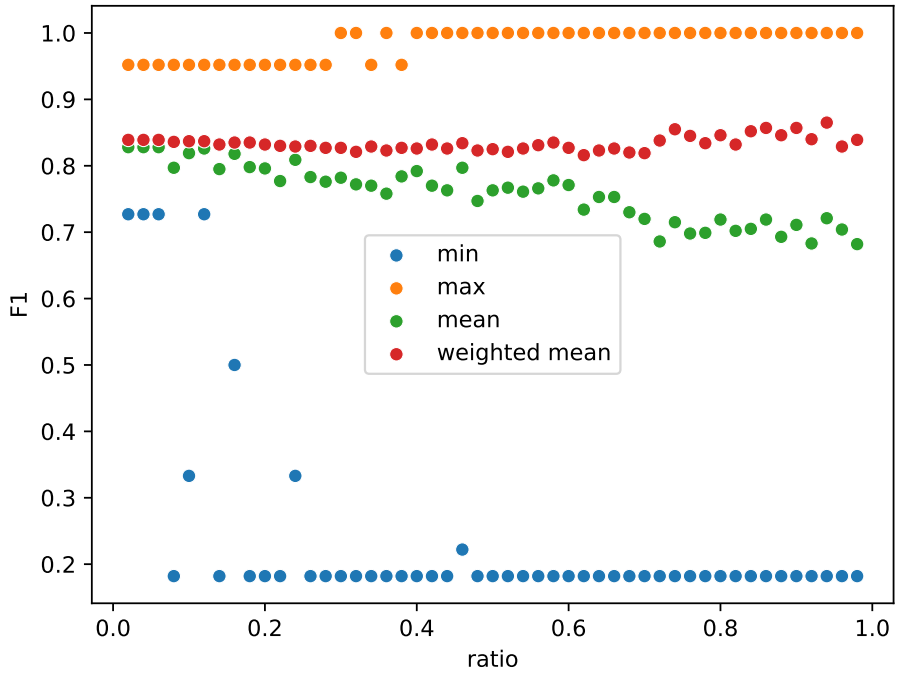

Abbildung 6.9: Evaluation der Verbindungssuche bei zunehmender Löschung von Linien; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,58; `similarity_threshold`: 0,12; git-tag: `eval-stage2-v22`

Diagramm	Präzision	Ausbeute	F1
`BBB`	1,00	0,67	0,80
`TM_A`	1,00	0,13	0,22
`TM_P`	0,86	0,32	0,47
`TM_U`	0,75	0,90	0,82
`TS`	1,00	1,00	1,00
`MS`	1,00	1,00	1,00

Tabelle 6.4: Evaluation der Verbindungssuche ohne *graph matching*; `epsilon`: 1,0; `max_iterations`: 0; `text_similarity_threshold`: 0,58; `similarity_threshold`: 0,12; git-tag: `eval-stage2-v22`

minimaler, maximaler und durchschnittlicher F1-Score für verschiedene Werte des Parameters dargestellt. Es wird 0,68 gewählt, da für diesen Wert sowohl hohes Minimum als auch hoher Durchschnitt erreicht werden.

Mit dem gewählten Parameter kann dann die Verwendung der angepassten Jaccard-Funktion in der Verbindungssuche evaluiert werden. Tabelle 6.5 zeigt die Ergebnisse, diese werden mit denen in Unterabschnitt 5.2.6 verglichen. Für `BBB` und `TM_A` ist die Qualität gestiegen. Für `TM_P`, `TM_U` und `MS` ist die Qualität gesunken. Für `TS` ist die

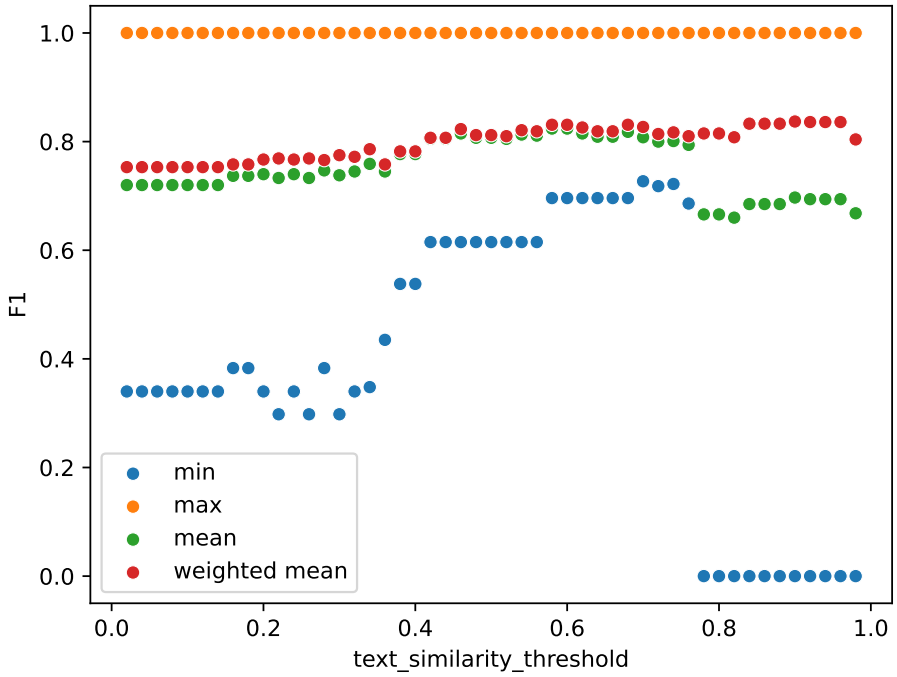

Abbildung 6.10: Untersuchung Parameter `text_similarity_threshold` für Verbindungssuche mit angepasster Jaccard-Funktion;
`epsilon`: 1,0; `max_iterations`: 100; `similarity_threshold`: 0,12; git-tag: `eval-stage2-v23`

Qualität gleich geblieben. Im Schnitt ist die Verwendung der Levenshtein-Distanz geringfügig besser. Die Art, wie die angepasste Jaccard-Funktion Groß- und Kleinschreibung behandelt, ist vermutlich ein Faktor für geringere Präzision bei einigen der Diagramme. Gleichzeitig ist diese Art der Behandlung aber semantisch sinnvoller, da zum Beispiel Abkürzungen in Schreibweise 'ABC' und 'abc' als ähnlich erkannt werden. Diese Eigenschaft wird gegenüber dem geringfügigen Qualitätsverlust bevorzugt. Daher wird die angepasste Jaccard-Funktion im Weiteren für die Verbindungssuche verwendet. Die hier dargestellten Ergebnisse sind somit die finalen Ergebnisse der Verbindungssuche.

6.2.5 Erweitertes Codemodell

Um eine größere Anzahl an Linien zwischen Boxen mit dem Modell abgleichen zu können, wurde das Codemodell erweitert. Die Erweiterung ist in Abschnitt 2.5 beschrieben. Da nur das Codemodell verändert wird, sind nur die Diagramme `TM_P` und `TM_U` davon betroffen. Die Auswirkungen der Erweiterung auf die Verbindungssuche werden in Tabelle 6.6 dargestellt. Für `TM_P` und `TM_U` steigt die Qualität stark an.

Diagramm	Präzision	Ausbeute	F1
BBB	1,00	1,00	1,00
TM_A	1,00	0,75	0,86
TM_P	0,88	0,68	0,77
TM_U	0,62	0,80	0,70
TS	0,80	0,67	0,73
MS	0,82	0,90	0,86

Tabelle 6.5: Evaluation der Verbindungssuche mit angepasster Jaccard-Funktion; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,68; `similarity_threshold`: 0,12; git-tag: `eval-stage2-v23`

Diagramm	Präzision	Ausbeute	F1
TM_P	0,89	0,73	0,80
TM_U	0,69	0,90	0,80

Tabelle 6.6: Evaluation der Verbindungssuche mit erweitertem Codemodell; `similarity_threshold_architecture`: 0,6; `similarity_threshold_code`: 0,8; `match_threshold`: 0,05; `match_difference`: 0,05; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,68; `similarity_threshold`: 0,12; git-tag: `eval-stage3-v7`

6.3 Inkonsistenz-Erkennung

Eine Evaluation der Inkonsistenz-Erkennung für sich alleine ist bei dem vorgestellten Ansatz nicht sinnvoll. Die vom Ansatz anfänglich verwendeten Konsistenz-Regeln basieren auf den Regeln, die bei der Erstellung des Goldstandards verwendet wurden. Somit erkennt der Ansatz bei Verwendung der Verbindungen im Goldstandard exakt alle Inkonsistenzen des Goldstandards. Für die weitere Evaluation werden daher alle drei Schritte gemeinsam evaluiert. Dadurch können die Auswirkungen von Fehlern in den vorherigen Schritten auf die Inkonsistenz-Erkennung evaluiert werden.

Somit sind auch die ersten zwei Forschungsfragen zu der Inkonsistenz-Erkennung hinfällig. Unabhängig von der Art der Inkonsistenzen werden bei Verwendung der Verbindungen im Goldstandard alle Inkonsistenzen gefunden.

6.3.1 Wahl der Parameter

Wie bereits in vorherigen Abschnitten geschildert, hat die Qualität der Verbindungssuche eine Auswirkung auf die Qualität der Inkonsistenz-Erkennung. Da die Parameter der Verbindungssuche so gewählt sind, dass eine möglichst hohe Qualität erreicht wird, sollten diese Parameter unter Betrachtung der Inkonsistenz-Erkennung geeignet sein. Dies wird in diesem Abschnitt evaluiert. Die folgenden Abbildungen 6.11 bis 6.13 zeigen die Ergebnisse der Evaluation der Parameter der Verbindungssuche. Es ist zu erkennen, dass

die bisher gewählten Parameter für die Inkonsistenz-Erkennung geeignet sind, wenn auch nicht unbedingt optimal. Für `similarity_threshold` ist der Wert 0,06 besser geeignet, vermutlich ist dieser Wert für die angepasste Jaccard-Funktion passender.

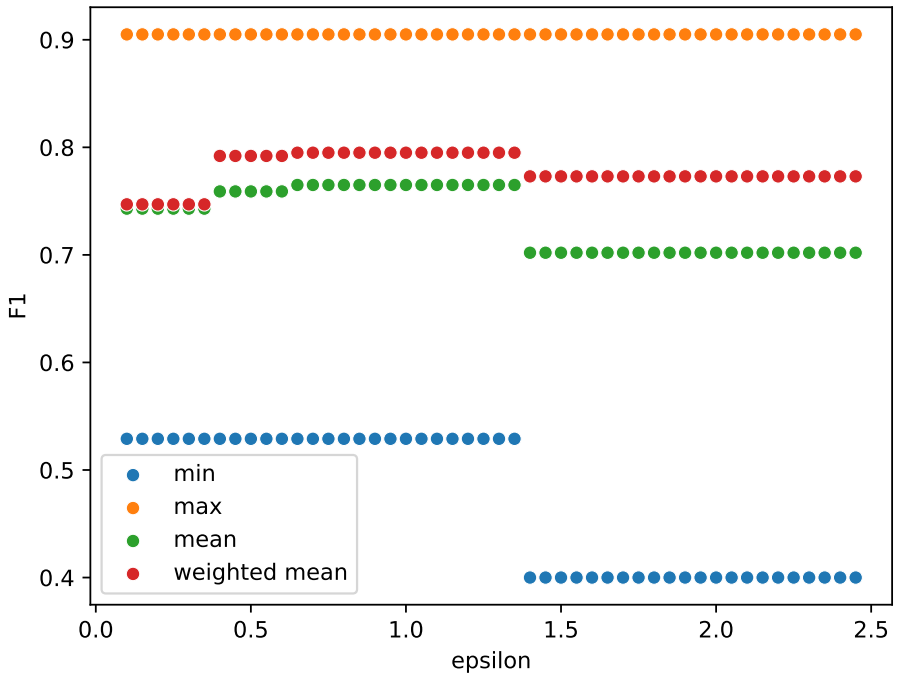

Abbildung 6.11: Untersuchung Parameter `epsilon` für Verbindungssuche im der Kontext Inkonsistenz-Erkennung; `max_iterations`: 100; `text_similarity_threshold`: 0,68; `similarity_threshold`: 0,12; git-tag: `eval-stage3-v7`

In Tabelle 6.7 sind die Ergebnisse der Inkonsistenz-Erkennung mit den finalen Parametern dargestellt. Die Qualität ist für alle Diagramme gestiegen oder gleich geblieben.

6.3.2 Betrachtung der Anwendung auf ein reales Diagramm

Im Folgenden werden die gefundenen Inkonsistenzen am Beispiel-Diagramm aus der Einleitung dargestellt. Abbildung 6.14 zeigt ein Architekturdiagramm des Projektes BigBlueButton [6]. Das Diagramm sowie ein Code- und Architekturmodell wurden an den Ansatz übergeben. Alle gefundenen Inkonsistenzen wurden behoben, das Ergebnis ist in Abbildung 6.15 dargestellt. In der Abbildung sind alle geänderten Elemente rot markiert.

Der größte Anteil der Änderungen betrifft die Namen der Boxen. Im ursprünglichen Diagramm nutzen die Namen keine Großbuchstaben, in der geänderten Version werden die Namen des Architekturmodells verwendet. Das Nutzen einer abweichenden Namenskonvention im Diagramm könnte hier eine bewusste Entscheidung sein. Eine weitere Kategorie

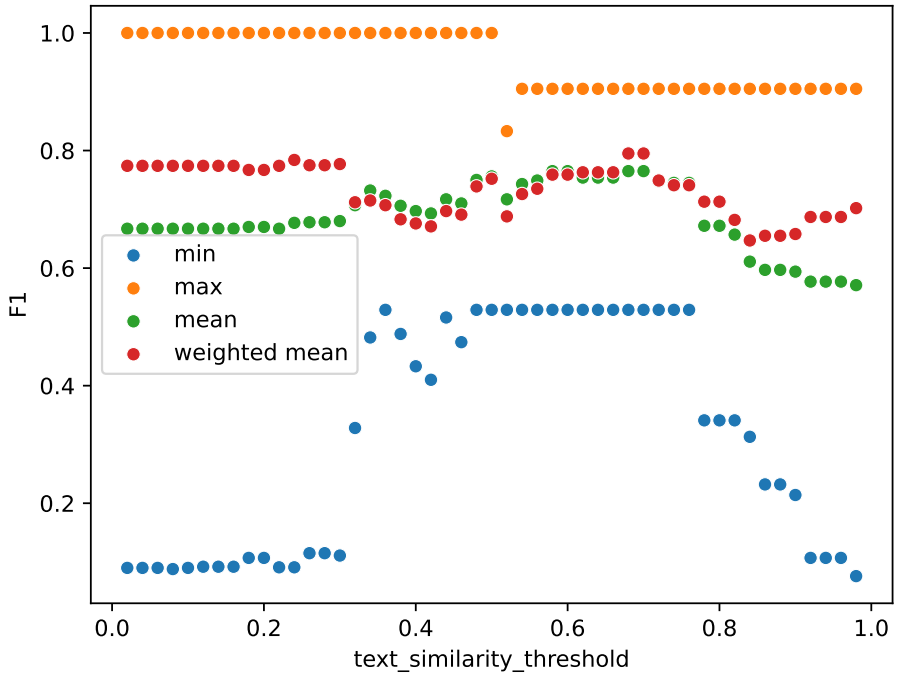

Abbildung 6.12: Untersuchung Parameter `text_similarity_threshold` für Verbindungssuche im der Kontext Inkonsistenz-Erkennung;
`epsilon`: 1,0; `max_iterations`: 100; `similarity_threshold`: 0,12; git-tag: `eval-stage3-v7`

Diagramm	Präzision	Ausbeute	F1
BBB	1,00	1,00	1,00
TM_A	0,77	0,77	0,77
TM_P	0,71	0,91	0,80
TM_U	0,74	0,85	0,79
TS	0,60	0,47	0,53
MS	0,79	0,73	0,76

Tabelle 6.7: Evaluation der Inkonsistenz-Erkennung mit finalen Parametern;
`similarity_threshold_architecture`: 0,6; `similarity_threshold_code`: 0,8; `match_threshold`: 0,05; `match_difference`: 0,05; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,68; `similarity_threshold`: 0,06; git-tag: `eval-stage3-v6`

von Änderungen sind geänderte Linienrichtungen. Sowohl die Linien mit zwei Pfeilen, als auch die Linien zu 'HTML5 Client' und 'HTML5 Server', sowie die Linie zwischen 'Presentation Conversion' und 'BBB web' stimmen nicht mit den im Architekturmodell definierten Abhängigkeiten überein. Die Box '3rd party' wird aus dem Diagramm entfernt, da es keine Entsprechung im Architekturmodell gibt. Dies hängt damit zusammen, dass

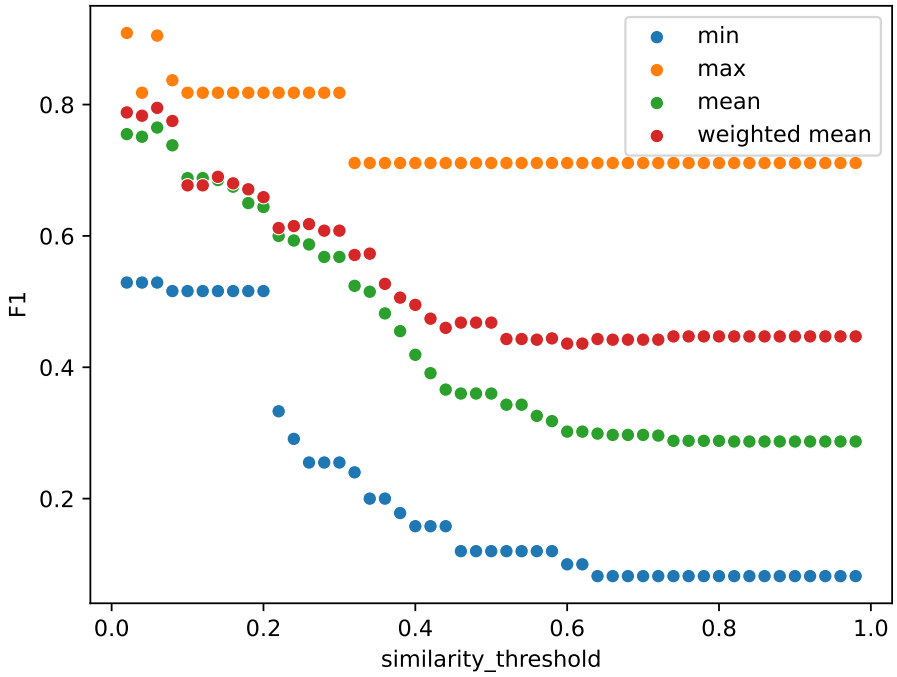

Abbildung 6.13: Untersuchung Parameter `similarity_threshold` für Verbindungssuche im der Kontext Inkonsistenz-Erkennung; `epsilon`: 1,0; `max_iterations`: 100; `text_similarity_threshold`: 0,68; git-tag: `eval-stage3-v7`

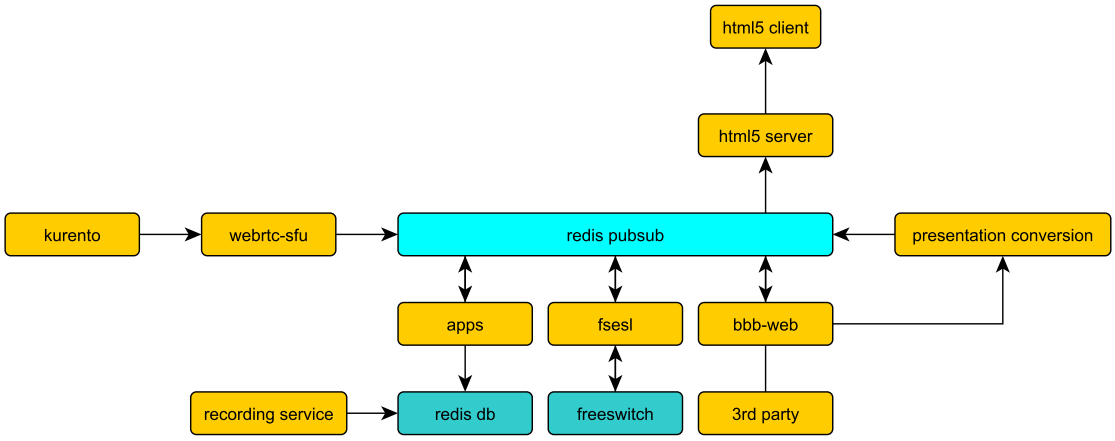

Abbildung 6.14: Wiederholung der Architekturübersicht von BigBlueButton [6]

die Box stellvertretend für Komponenten steht, welche nicht Teil des Projektes und somit der Architektur sind.

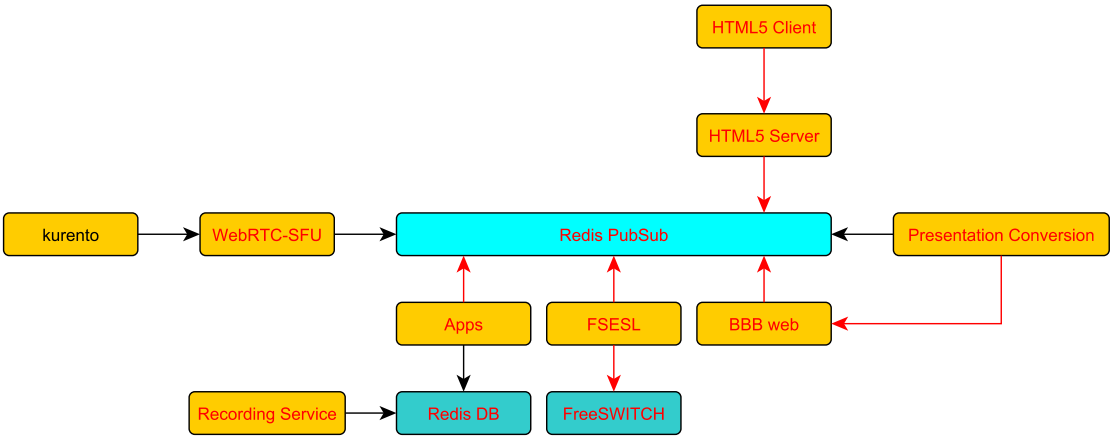

Abbildung 6.15: Angepasste Architekturübersicht von BigBlueButton [6]

Die geschilderten Änderungen werden vom Ansatz in insgesamt 22 Inkonsistenzen präsentiert. Mit den in Unterabschnitt 5.3.2 beschriebenen Methoden werden diese in 8 erweiterten Inkonsistenzen zusammengefasst. Alle Änderungen bezüglich der Benennung werden als eine einzige Inkonsistenz präsentiert. Sollte die verwendete Namenskonvention im Diagramm bewusst von der im Modell abweichen, so muss nur noch eine Inkonsistenz ignoriert werden. Alle unerwartet aus 'redis pubsub' ausgehenden Linien werden in einer Gruppe präsentiert. Statt zweier Inkonsistenzen je invertierter Linie wird nur noch eine Inkonsistenz präsentiert.

7 Fazit

In dieser Arbeit wurde ein Ansatz zur automatischen Typisierung und Konsistenz-Überprüfung von informellen Diagrammen dargestellt. Der Ansatz sucht Nachverfolgbarkeitsverbindungen zwischen Diagrammen in einer einfachen Boxen-und-Linien-Form und Architektur- sowie Codemodellen. Dazu wird *graph matching* verwendet, welches auch auf Basis der Struktur anstelle ausschließlich auf Basis von Texten Verbindungen erkennen soll. Ob *graph matching* sinnvoll ist, hängt dabei vom Diagramm ab und ob die Inkonsistenzen stärker in der Struktur oder in den Bezeichnern präsent sind. Für die Mehrzahl der betrachteten Diagramme bietet *graph matching*, gemessen am F1-Score, ein Vorteil gegenüber reinen Textvergleichen.

Bei der Entwicklung wurden sowohl synthetische als auch reale Diagramme verwendet. Für die meisten betrachteten Diagramme gelingt es dem Ansatz, das im Diagramm dargestellte Modell zu identifizieren. Ein Großteil der in den Diagrammen vorhandenen Inkonsistenzen werden erkannt und ein durchschnittlicher F1-Score von 77,5% erreicht. Gleichzeitig werden aber auch im Goldstandard nicht erwartete Inkonsistenzen vom Ansatz erkannt. Diese Inkonsistenzen können weiter hilfreich bei der Konsistenzhaltung von Diagrammen sein, wie sehr diese helfen kann, in zukünftigen Arbeiten untersucht werden. Eine Evaluation der Nutzbarkeit des Ansatzes für reale Projekte ist hierfür nötig.

Bei der Entwicklung des Ansatzes wurden synthetische Diagramme eingesetzt. Vorteile dieser sind Verfügbarkeit und Kontrolle über die darin enthaltenen Inkonsistenzen. Des Weiteren dienten diese dazu, systematisch Klassen von Änderungen an Modellen und Diagrammen zu betrachten und somit eine allgemeine Behandlung dieser zu ermöglichen. Der Unterschied der gemessenen Metriken bei realen und synthetische Diagrammen zeigt, dass die generierten Diagramme grundlegend andere Eigenschaften als reale Diagramme aufweisen.

Für weiterführende Arbeiten zeichnen sich, neben der Evaluation der Nutzbarkeit und einer Verbesserung der Verbindungserkennung, hauptsächlich zwei Gebiete ab. Das erste Gebiet ist die Erweiterung des Boxen-und-Linien-Formates, um die Eigenschaften realer Diagramme besser abbilden zu können. Dazu gehört, bereits bei der Erkennung von Diagrammen verschiedene Arten von Boxen, Linien und Texten zu unterscheiden. Dadurch könnten zum Beispiel beschreibende Texte von Bezeichnern getrennt werden. Das zweite Gebiet ist das Einbringen von Erkenntnissen aus realen Anwendungen in die drei Stufen des Ansatzes, insbesondere in die Inkonsistenz-Erkennung. Dies würde die allgemeine Anwendbarkeit schwächen, könnte aber die Nutzbarkeit verbessern. Eine Vielzahl spezialisierter Regeln, zum Beispiel zur Erkennung fehlender Boxen, sowie verfeinerte

Inkonsistenz-Typen sind hier denkbar. Auch bei der Verbindungssuche könnte Wissen über Konventionen in Programmiersprachen und Projekten verwendet werden.

Zusätzlich ist es denkbar, den Ansatz für weitere Typen von Diagrammen zu nutzen. Ein weiteres, die Struktur abbildendes Diagramm ist das Schichtendiagramm, und auch Diagramme, die Abläufe abbilden, sind denkbare Anwendungen. Dazu ist primär eine Erweiterung der vorhandenen, beziehungsweise die Erstellung neuer Modellarten notwendig.

Literatur

[1] *About the Unified Modeling Language Specification Version 2.5.1*. URL: https://www.omg.org/spec/UML/2.5.1/About-UML/ (besucht am 28. 05. 2023).

[2] Craig Anslow u. a. „Visualizing the Word Structure of Java Class Names“. In: *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA Companion '08. New York, NY, USA: Association for Computing Machinery, Okt. 2008, S. 777–778. ISBN: 978-1-60558-220-7. DOI: 10.1145/1449814.1449857. URL: https://dl.acm.org/doi/10.1145/1449814.1449857 (besucht am 19. 05. 2023).

[3] Author. *KIT - MCSE - Research - Projects - ArDoCo*. Text. Apr. 2023. URL: https://mcse.kastel.kit.edu/Projects_ArDoCo.php (besucht am 15. 07. 2023).

[4] Sebastian Baltes und Stephan Diehl. „Sketches and Diagrams in Practice“. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 2014, S. 530–541. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635891. URL: https://dl.acm.org/doi/10.1145/2635868.2635891 (besucht am 12. 05. 2023).

[5] *Benchmark*. ArDoCo. März 2023. URL: https://github.com/ArDoCo/Benchmark (besucht am 09. 06. 2023).

[6] *BigBlueButton : Architecture*. März 2023. URL: http://web.archive.org/web/20230315102607/https://docs.bigbluebutton.org/2.4/architecture.html (besucht am 27. 05. 2023).

[7] D. Dig und R. Johnson. „The Role of Refactorings in API Evolution“. In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*. Sep. 2005, S. 389–398. DOI: 10.1109/ICSM.2005.90.

[8] Wei Ding u. a. „How Do Open Source Communities Document Software Architecture: An Exploratory Survey“. In: *2014 19th International Conference on Engineering of Complex Computer Systems*. Aug. 2014, S. 136–145. DOI: 10.1109/ICECCS.2014.26.

[9] Wael H. Gomaa und Aly A. Fahmy. „A Survey of Text Similarity Approaches“. In: *International Journal of Computer Applications* 68.13 (Apr. 2013), S. 13–18. URL: https://www.ijcaonline.org/archives/volume68/number13/11638-7118 (besucht am 16. 09. 2023).

[10] Orlena Gotel u. a. „Traceability Fundamentals“. In: *Software and Systems Traceability*. Hrsg. von Jane Cleland-Huang, Orlena Gotel und Andrea Zisman. London: Springer, 2012, S. 3–22. ISBN: 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5_1. URL: https://doi.org/10.1007/978-1-4471-2239-5_1 (besucht am 09. 06. 2023).

[11] Birgit Grammel, Stefan Kastenholz und Konrad Voigt. „Model Matching for Trace Link Generation in Model-Driven Software Development“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Robert B. France u. a. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, S. 609–625. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_39.

[12] Patrick A. V. Hall und Geoff R. Dowling. „Approximate String Matching“. In: *ACM Computing Surveys* 12.4 (Dez. 1980), S. 381–402. ISSN: 0360-0300. DOI: 10.1145/356827.356830. URL: https://dl.acm.org/doi/10.1145/356827.356830 (besucht am 02. 11. 2023).

[13] Kamel Eddine Heraguemi, Abdelatif Abid und Abdelkrim Amirat. „Software Architecture Matching Based on Similarity Flooding Algorithm“. In: *Proceedings of MISC 2012 - 2nd International Symposium on Modelling and Implementation of Complex Systems*.

[14] Steven A. Hicks u. a. „On Evaluation Metrics for Medical Applications of Artificial Intelligence“. In: *Scientific Reports* 12 (Apr. 2022), S. 5979. ISSN: 2045-2322. DOI: 10.1038/s41598-022-09954-8. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8993826/ (besucht am 04. 11. 2023).

[15] Nicholas J. Higham. „1. Principles of Finite Precision Computation“. In: *Accuracy and Stability of Numerical Algorithms*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2002, S. 1–33. ISBN: 978-0-89871-521-7. DOI: 10.1137/1.9780898718027.ch1. URL: https://epubs.siam.org/doi/10.1137/1.9780898718027.ch1 (besucht am 10. 09. 2023).

[16] Robbert Jongeling u. a. „From Informal Architecture Diagrams to Flexible Blended Models“. In: *Software Architecture*. Hrsg. von Ilias Gerostathopoulos u. a. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, S. 143–158. ISBN: 978-3-031-16697-6. DOI: 10.1007/978-3-031-16697-6_10.

[17] Jan Keim u. a. „Trace Link Recovery for Software Architecture Documentation“. In: *Software Architecture: 15th European Conference, ECSA 2021, Virtual Event, Sweden, September 13-17, 2021, Proceedings. Ed.: S. Biffl*. 2021, S. 101. ISBN: 978-1-00-013839-9. DOI: 10.1007/978-3-030-86044-8_7. URL: https://publikationen.bibliothek.kit.edu/1000138399 (besucht am 09. 05. 2023).

[18] Markus Kleffmann u. a. „Establishing and Navigating Trace Links between Elements of Informal Diagram Sketches“. In: *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*. Mai 2015, S. 1–7. DOI: 10.1109/SST.2015.8.

[19] Bin Luo und E.R. Hancock. „Structural Graph Matching Using the EM Algorithm and Singular Value Decomposition“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.10 (Okt. 2001), S. 1120–1136. ISSN: 1939-3539. DOI: 10.1109/34.954602.

[20] A.J. Malton und R.C. Holt. „Boxology of NBA and TA: A Basis for Understanding Software Architecture“. In: *12th Working Conference on Reverse Engineering (WCRE'05)*. Nov. 2005, 9 pp.–195. DOI: 10.1109/WCRE.2005.10.

[21] S. Melnik, H. Garcia-Molina und E. Rahm. „Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching“. In: *Proceedings 18th International Conference on Data Engineering*. Feb. 2002, S. 117–128. DOI: 10.1109/ICDE.2002.994702.

[22] Sergey Melnik, Hector Garcia-Molina und Erhard Rahm. „Similarity Flooding: A Versatile Graph Matching Algorithm (Extended Technical Report)“. In: URL: https://www.researchgate.net/profile/Erhard-Rahm/publication/279509283_Similarity_Flooding_A_Versatile_Graph_Matching_Algorithm_Extended_Technical_Report/links/568d68e608aef987e565efbf/Similarity-Flooding-A-Versatile-Graph-Matching-Algorithm-Extended-Technical-Report.pdf (besucht am 26. 07. 2023).

[23] Annibale Panichella u. a. „When and How Using Structural Information to Improve IR-Based Traceability Recovery“. In: *2013 17th European Conference on Software Maintenance and Reengineering*. März 2013, S. 199–208. DOI: 10.1109/CSMR.2013.29.

[24] Robert Pienta u. a. „MAGE: Matching Approximate Patterns in Richly-Attributed Graphs“. In: *2014 IEEE International Conference on Big Data (Big Data)*. Okt. 2014, S. 585–590. DOI: 10.1109/BigData.2014.7004278.

[25] Ralf H. Reussner u. a. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. ISBN: 9780262034760.

[26] *TEAMMATES - Design*. URL: https://teammates.github.io/teammates/design.html (besucht am 24. 07. 2023).

[27] Tobias Telge. *Automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext*. 2023. DOI: 10.5445/IR/1000157372. URL: https://publikationen.bibliothek.kit.edu/1000157372 (besucht am 09. 05. 2023).

[28] Jóakim von Kistowski u. a. „TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research“. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Sep. 2018, S. 223–236. DOI: 10.1109/MASCOTS.2018.00030.

[29] William E. Winkler. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. Techn. Ber. 1990. URL: https://eric.ed.gov/?id=ED325505 (besucht am 02. 11. 2023).

[30] Junchi Yan u. a. „A Short Survey of Recent Advances in Graph Matching“. In: *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval*. ICMR '16. New York, NY, USA: Association for Computing Machinery, Juni 2016, S. 167–174. ISBN: 978-1-4503-4359-6. DOI: 10.1145/2911996.2912035. URL: https://dl.acm.org/doi/10.1145/2911996.2912035 (besucht am 13. 05. 2023).