FH Aachen
Fachbereich Elektrotechnik und Informationstechnik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Bachelorarbeit

Zur Erlangung des akademischen Grades
Bachelor of Science

Automatisierte Generierung von Migrationsskripten für CIM-Modelle bei RDF-Schema-Änderungen

Eingereicht von: Finn Tarnowsky
Matrikelnummer: 3566415

Datum: 13. Februar 2026

Betreuer: Prof. Dr. rer. nat. Heinrich Faßbender
Zweitprüfer: Michael Lomb (B.Sc.), SOPTIM AG

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.
Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Nachweis versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Aachen, 13. Februar 2026

Finn Tarnowsky

Abkürzungsverzeichnis

CIM Common Information Model

CGMES Common Grid Model Exchange Specification

ENTSO-E European Network of Transmission System Operators for Electricity

EPRI Electric Power Research Institute

IEC International Electrotechnical Commission

IRI Internationalized Resource Identifier

RDF Resource Description Framework

RDFS RDF Schema

REST Representational State Transfer

SHACL Shapes Constraint Language

SPARQL SPARQL Protocol and RDF Query Language

W3C World Wide Web Consortium

XSD XML Schema Definition

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Entwicklung des Common Information Model (CIM) begann in den frühen 1990er-Jahren im Rahmen des Control Center Application Programming Interface Projekts des Electric Power Research Institute (EPRI). Dessen Ziel war es, ein gemeinsames semantisches Modell für die Energiewirtschaft zu schaffen und dadurch den standardisierten Datenaustausch zwischen verschiedenen Netzbetreibern und IT-Systemen zu ermöglichen. Seit 1996 wird der CIM Standard von dem Technical Committee 57 der International Electrotechnical Commission (IEC) betreut und besteht aus mehreren kleineren Profilen, die als IEC Spezifikationen veröffentlicht werden. [1], [2]

Der fortschreitende technische Wandel in der Energiewirtschaft setzt jedoch voraus, dass diese Profile kontinuierlich weiterentwickelt und an den neuen Stand der Technik angepasst werden. Um den spezifischen Anforderungen der Stromübertragung und -verteilung gerecht zu werden, entwickelte das European Network of Transmission System Operators for Electricity auf Basis von CIM die Common Grid Model Exchange Specification (CGMES). Diese wird ebenfalls kontinuierlich weiterentwickelt, wobei verschiedene Versionen parallel in der Energiewirtschaft im Einsatz sind. [3] Hinzu kommen zahlreiche aufbauende Modelle der einzelnen Netzbetreiber selbst.

Die fortlaufende Weiterentwicklung des CIM und CGMES macht es notwendig, die Instanzdaten regelmäßig an die neuen Modelle anzupassen. Da für die Migration von CIM-Daten bisher keine unterstützenden Tools zur Verfügung stehen, müssen die dafür benötigten Migrationsskripte in der Regel manuell erstellt werden. Dieses Verfahren ist aufgrund der Größe und Komplexität der Schemata sowohl fehleranfällig als auch zeitaufwändig.

Die vorliegende Arbeit entsteht im Auftrag der SOPTIM AG und soll eines ihrer Tools, den RDF-Architect, weiterentwickeln. Ziel ist es, den Prozess der Schemamigration für CIM-konforme Datenmodelle weitestgehend zu automatisieren und damit schneller und zuverlässiger zu machen.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, den bestehenden RDF-Architect um eine Funktionalität zu erweitern, die den Anwender bei der Schemamigration von CIM-konformen Datenmodellen unterstützt. Dazu werden zunächst die möglichen Änderungen zwischen zwei Versionen eines CIM-Schemas systematisch analysiert und die jeweils erforderlichen Migrationsschritte identifiziert.

Eine besondere Herausforderung stellt die Erkennung von Umbenennungen dar, da diese nicht unmittelbar aus dem Schemavergleich hervorgehen. Daher werden verschiedene Methoden zur automatischen Erkennung von Umbenennungen untersucht und evaluiert. Hierbei kommen sowohl stringbasierte Vergleichsverfahren als auch die Analyse von Strukturdaten zum Einsatz, um Umbenennungen von tatsächlichen gelöschten und neu hinzugefügten Ressourcen heuristisch zu unterscheiden.

Darauf aufbauend wird ein nutzerorientierter UI-Fluss konzipiert, der die erkannten Änderungen visualisiert und dem Anwender Entscheidungs- und Konfigurationsmöglichkeiten bietet. Die entwickelte Funktionalität erzeugt schließlich auf Basis der erkannten Änderungen und der Nutzereingaben ein SPARQL-Skript, das die Migration der Instanzdaten ermöglicht.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in fünf Kapitel.

Kapitel 2 legt die theoretischen und technischen Grundlagen für die Arbeit. Es beschreibt die grundlegenden Konzepte der Modellierung von CIM-Daten, insbesondere die Struktur der verschiedenen Arten von Ressourcen. Das Kapitel schließt mit einer Einführung in SHACL, SPARQL und verschiedene Stringvergleichsmethoden, die für die Umbenennungserkennung relevant sind.

In Kapitel 3 wird der RDF-Architect vorgestellt. Es werden die bestehenden Kernfunktionalitäten der Anwendung beschrieben und gesondert auf den bereits implementierten Schemavergleich eingegangen. Anschließend werden die technischen Details und bisher verwendeten Konventionen des Backends und Frontends erläutert, um die Grundlage für die Erweiterung der Anwendung zu schaffen.

Kapitel 4 befasst sich mit der Konzeption der Schemamigration. Neben der Analyse der möglichen Schemaänderungen werden verschiedene Ansätze zur Umbenennungserkennung untersucht. Dabei werden sowohl bestehende Verfahren analysiert, als auch eine eigene Lösung basierend auf Stringvergleichsmethoden und Strukturdaten entwickelt. Das Kapitel beschreibt zudem das Design des User Interfaces und die

dafür nötige Verwaltung des Migrationszustandes.

Kapitel 5 erläutert die Implementierung des entwickelten Ansatzes im Front- und Backend. Dies umfasst die Datenmodelle, die Erkennung semantischer Änderungen, die Implementierung der Umbenennungserkennung unter Berücksichtigung von Vererbungsbeziehungen sowie die Generierung der SPARQL-Migrationsskripte.

Kapitel 6 fasst die Ergebnisse der Arbeit zusammen und bewertet die Zielerreichung. Es diskutiert die Limitationen des entwickelten Ansatzes und der automatisierten Schemamigration generell, und gibt einen Ausblick auf mögliche Erweiterungen.

2 Grundlagen und Technologien

Um den Ablauf einer Datenmigration zwischen CIM-Schemata nachvollziehen zu können, ist ein grundlegendes Verständnis von RDF und CIM notwendig. Dieses Kapitel gibt einen Überblick über die Grundlagen und stellt die verwendeten Technologien vor.

Abschnitt 2.1 erläutert die Grundlagen von RDF und wie Daten in diesem Format dargestellt werden können. Außerdem wird SPARQL als Querysprache für RDF vorgestellt. In Abschnitt 2.2 wird das CIM beschrieben, welches in der Energiewirtschaft als Standard für die Modellierung von Daten dient. In Abschnitt 2.3 wird erläutert, wie SHACL genutzt werden kann, um RDF-Graphen zu validieren. Anschließend gibt Abschnitt 2.4 einen Überblick über verschiedene Algorithmen zur Bestimmung der Ähnlichkeit von Zeichenketten, welche im Kontext der Umbenennungserkennung verwendet werden können. Abschließend werden in Abschnitt 2.5 die in dieser Arbeit verwendeten Technologien vorgestellt.

2.1 Resource Description Framework

Das Resource Description Framework (RDF) beschreibt eine Methode Graphdaten im Web auszutauschen. Es wurde ursprünglich vom World Wide Web Consortium (W3C) entwickelt, um Metadaten im Internet auszutauschen und bildet seitdem einen Grundstein des Semantic Web [4]. RDF kann genutzt werden, um beliebige Aussagen, sogenannte Statements, über Dinge aus verschiedensten Domänen zu treffen. Diese Dinge werden als Ressourcen bezeichnet [5].

In RDF werden Statements durch Tripel abgebildet und bestehen immer aus jeweils einem Subjekt, Prädikat und Objekt. Das Subjekt gibt die Ressource an, die beschrieben wird, das Prädikat, welche Eigenschaft der Ressource beschrieben wird, und das Objekt weist der Eigenschaft einen bestimmten Wert zu. So könnte man beispielsweise den Satz „Die Person heißt Max Mustermann“ als Tripel mit der Person als Subjekt, einem „Name“-Prädikat und „Max Mustermann“ als Objekt ausdrücken. Weitere Aussagen über die Person können daraufhin einfach angehängt werden. [6]

Die Bestandteile eines Tripels, auch Nodes genannt, können in verschiedenen Formen auftreten. Subjekte und Prädikate sind immer als ein Internationalized Resource Identifier (IRI) angegeben. Diese IRIs sind im Aufbau ähnlich zu URLs, müssen

jedoch nicht auf eine Webseite verweisen, sondern dienen primär der eindeutigen Identifikation von Ressourcen und Eigenschaften. [6]

IRIs bestehen immer aus zwei Teilen: dem Präfix und dem eigentlichen Namen. Der Präfix, oder auch Namespace, gibt meistens an, wann und von wem eine IRI registriert wurde. Alle Prädikate, die von W3C im Rahmen von RDF registriert wurden, tragen beispielsweise den Präfix `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. Da das Ausschreiben von Präfixen jedoch schnell mühsam werden kann und Dateien unübersichtlich macht, werden Präfixe häufig abgekürzt. So wird `http://www.w3.org/1999/02/22-rdf-syntax-ns#` z.B. durch `rdf:` ersetzt. Diese Abkürzungen lassen sich pro Graph definieren, für häufig verwendete Präfixe haben sich jedoch einige Standardabkürzungen etabliert. [7], [8]

IRIs, die im Rahmen von Beispielen für diese Arbeit definiert werden, werden mit dem Präfix `http://example.org` angelegt, welcher durch `ex:` abgekürzt wird.

Objekte können sowohl als IRI angegeben werden, wenn sie auf eine andere Ressource verweisen, als auch einen literalen Wert enthalten. Literalen können weitere Metainformationen angehängt werden, wie beispielsweise ihr Datentyp oder eine Sprachangabe. Wie genau diese Metainformationen angegeben werden, hängt von dem Serialisierungsformat ab, in dem der Graph dargestellt wird. Unterabschnitt 2.1.2 enthält ein Beispiel für die Serialisierung im Turtle-Format. [9]

Letztlich können Objekte statt als IRI oder Literal auch in Form einer Blank-Node auftreten. Diese Blank-Nodes sind anonyme und nicht benannte Ressourcen. Durch ihre fehlende IRI können sie nirgendwo anders referenziert werden, bieten im Vergleich zu normalen Nodes jedoch den Vorteil, dass sie mehrere Eigenschaften zusammenfassen können. [10] Da Blank-Nodes in den verwendeten CIM-Schemata jedoch selten auftreten, werden sie in dieser Arbeit nicht weiter behandelt.

2.1.1 RDF Schema

RDF Schema (RDFS) wurde ebenfalls von W3C entwickelt und erweitert das Vokabular von RDF um eine Vielzahl grundlegender Modellierungskonzepte und bietet so die Möglichkeit, eigene Klassenmodelle mittels RDF-Graphen zu erstellen. Zugehörige IRIs sind im Namespace `http://www.w3.org/2000/01/rdf-schema#` enthalten, welcher standardmäßig mit `rdfs:` abgekürzt wird. [11] Die konkreten RDFS-Konstrukte und ihre Verwendung im Kontext von CIM-Schemata werden in Unterabschnitt 2.2.3 detailliert erläutert.

2.1.2 Serialisierung

RDF kann in verschiedene Formate serialisiert werden, wobei in der vorliegenden Arbeit vorwiegend das Turtle-Format (Terse RDF Triple Language) verwendet wird, da es für Menschen am besten lesbar ist.

Tripel werden in Turtle durch eine einfache leerzeichengetrennte Aufzählung von Subjekt, Prädikat und Objekt dargestellt, wobei ein Tripel jeweils auf ein `.` endet. Mehrere Tripel mit dem gleichen Subjekt können in Turtle zusammengefasst werden, indem das Subjekt nur beim ersten Mal aufgeführt wird. Die darauf folgenden lassen das Subjekt entfallen und werden üblicherweise eingerückt. Diese Auflistung wird jeweils mit einem Semikolon getrennt und endet, wenn ein Tripel mit einem Punkt beendet wird, woraufhin das nächste Tripel wieder mit einem Subjekt beginnen muss. Zudem wird das häufig verwendete Prädikat `rdf:type` in Turtle meistens durch ein `a` abgekürzt. Datentypen werden in Turtle mittels eines doppelten Zirkumflex `^^` dem Literal angehängt. Sprachinformationen werden auf ähnliche Weise mit einem `@` angehängt. [12]

Code 2.1 zeigt ein beispielhaftes Turtle-Dokument, welches eine Person „Max Mustermann“ definiert.

```
ex:Max a ex:Person ;
    ex:name "Max Mustermann"@de ;
    ex:alter 25^^xsd:integer ;
    ex:wohnort ex:Musterstadt .
```

Code 2.1: Turtle Serialisierung mit Datentypen und Language Tags

Neben Turtle existieren noch viele weitere Formate, wie das in der Energiewirtschaft gängige RDF/XML, N-Triples und JSON-LD, die ebenfalls vom RDF-Architect unterstützt werden.

2.1.3 SPARQL

Die SPARQL Protocol and RDF Query Language (SPARQL) ist die von W3C empfohlene Query Sprache für RDF-Graphen und basiert auf dem Konzept des Pattern Matchings. SPARQL-Queries nutzen typischerweise eine `WHERE` Clause mit einem Triple-Pattern, welches genutzt werden kann, um ein Solution Set zu ermitteln. Die Nodes in diesem Triple-Pattern können entweder als eine IRI oder ein Literal angegeben werden, um das Solution Set einzugrenzen, oder als Variable. Variablen werden dabei immer mit einem `?` angeführt. Ein Solution Set

enthält eine Menge aller möglichen Variablenbelegungen, welche die angegebenen Triple-Patterns erfüllen. [13]

Das allgemeinste Triple-Pattern `?subject ?predicate ?object` würde beispielsweise alle Tripel des Graphen liefern, während das Pattern `ex:Max ?predicate ?object` nur die Tripel zurückgibt, die `ex:Max` als Subjekt haben.

Um komplexere Relation abzubilden, können mehrere Triple-Pattern verknüpft werden. Die Pattern werden dabei nacheinander durchlaufen, um die möglichen Belegungen einzugrenzen. [14] Code 2.2 zeigt, wie man die Großeltern von Max finden kann. Zuerst werden alle Tripel gesucht, welche die Eltern von Max definieren. Diese möglichen Belegungen von `?parent` können dann im zweiten Pattern genutzt werden, um Belegungen für `?grandparent` zu finden.

```
WHERE {
    ex:Max ex:childOf ?parent .
    ?parent ex:childOf ?grandparent .
}
```

Code 2.2: Komplexes Pattern

Queries

Um das Ergebnis eines Pattern Matches darzustellen, wird der SELECT Block genutzt. Code 2.3 zeigt, wie man Code 2.2 erweitern kann, um die Belegungen für `?grandparent` anzuzeigen. Dabei bleibt der Pattern Match unabhängig davon, welche Variablen im SELECT abgefragt werden. Die Belegungen für `?parent` sind also nicht beeinträchtigt, auch wenn sie von dem `SELECT` nicht abgefragt werden. [15]

```
SELECT ?grandparent
WHERE {
    ex:Max ex:childOf ?parent .
    ?parent ex:childOf ?grandparent .
}
```

Code 2.3: SELECT Beispiel

Updates

SPARQL kann nicht nur genutzt werden, um einen RDF-Graphen abzufragen, sondern kann diesen auch direkt bearbeiten. Dies geschieht über die Blöcke DELETE und INSERT. Ein potentieller `UPDATE` Block existiert in SPARQL nicht, da Tripel in RDF als 3-Tupel definiert und sind damit unveränderlich sind, weshalb einzelne Nodes eines Tripels mit SPARQL nicht direkt bearbeitet werden können. Stattdessen müssen die gesamten Tripel gelöscht und neu eingefügt werden. [16]

Code 2.4 zeigt, wie man mithilfe von `DELETE` und `INSERT` das Alter aller Personen, die 17 sind, auf 18 erhöhen kann.

```
DELETE {
    ?person ex:age 17
}
INSERT {
    ?person ex:age 18
}
WHERE {
    ?person ex:age 17
}
```

Code 2.4: Update Beispiel

In dem Beispiel ist zu erkennen, dass sich das gleiche Pattern im DELETE und WHERE Block wiederholt. Diese scheinbare Redundanz ist allerdings notwendig, da nicht alle durch den `WHERE` Block definierten Tripel zwingend gelöscht werden sollen. Dies ist insbesondere bei komplexeren `WHERE` Blöcken der Fall, dir mehrere Tripel Patterns benötigen.

2.2 Common Information Model

Das Common Information Model ist ein von der IEC gepflegtes Modell zu Beschreibung von elektrischen Energienetzen. CIM ist in den IEC-Normen 61970 und 61968 spezifiziert und wurde entwickelt, um die Interoperabilität zwischen verschiedenen Systemen und Organisationen im Energiesektor zu ermöglichen. [1]

CIM nutzt dabei RDF als technische Grundlage und erweitert dessen Vokabular um spezifische Modellierungskonzepte für Energiesysteme. Dafür gibt es die Struktur und Syntax von Klassen, Attributen, Assoziationen und Enums vor, und wie diese miteinander in Beziehung gesetzt werden können. Im Folgenden werden

Attribute, Assoziationen und Enum-Entries, die für Klassen definiert werden, als deren Properties bezeichnet.

IRIs, die im Rahmen von CIM definiert werden, haben in RDF-Graphen den Präfix `http://iec.ch/TC57/1999/rdf-schema-extensions-19990926#`, abgekürzt durch `cims:`. CIMS steht dabei für CIM-Schema.

2.2.1 ENTSO-E

Das European Network of Transmission System Operators for Electricity (ENTSO-E) ist der Verbund aller europäischen Übertragungsnetzbetreiber und wurde 2009 als Nachfolger der European Network Transmission Operators in Brüssel gegründet. Ziel des Verbunds ist es, die Zusammenarbeit zwischen den Übertragungsnetzbetreibern zu fördern und die Integration der europäischen Energiemärkte zu unterstützen. Diese bessere Zusammenarbeit soll der EU dabei helfen, mehr erneuerbare Energien zu integrieren und sicheren, nachhaltigen und erschwinglichen Strom bereitzustellen. [17]

Um diese Ziele zu erreichen, ist ein zuverlässiger und standardisierter Austausch von Daten zwischen den Übertragungsnetzbetreibern unabdingbar. ENTSO-E hat daher CGMES als verbindlichen Standard für den Datenaustausch etabliert und spielt eine tragende Rolle in deren Weiterentwicklung und Adoption.

2.2.2 CGMES

Die Common Grid Model Exchange Specification ist ein auf CIM basierendes Austauschformat für elektrische Netzmodelle, welches von ENTSO-E entwickelt wurde, um den sich weiterentwickelnden Anforderungen des europäischen Energiemarktes gerecht zu werden.

Während CIM das generische Modellierungsframework bereitstellt, definiert CGMES konkrete Profile, die zwischen verschiedenen Übertragungsnetzbetreibern ausgetauscht werden können. Die Profile sind thematisch gruppiert und repräsentieren verschiedene Aspekte eines Netzmodells. Beispielsweise enthält das „Equipment Profile“ Klassen für alle technischen Bauteile. während das „Steady State Hypothesis Profile“ Klassen für Zustandswerte bereit stellt. [18]

Die CGMES-Profile existieren dabei in verschiedenen Versionen. Die heute am häufigsten genutzten Versionen sind CGMES 2.4.15 (basierend auf CIM 16) und CGMES 3.0 (basierend auf CIM 17/18).

2.2.3 Arten von Ressourcen

Um einen standardisierten Austausch von Daten zu gewährleisten, ist für die verschiedenen Arten von Ressourcen im CIM eine feste Syntax und Struktur definiert. Dieser Abschnitt stellt diese Syntax der verschiedenen Ressourcentypen vor.

CIM-Packages

Packages werden in CIM-Modellen genutzt, um mehrere Klassen thematisch zu gruppieren. Packages werden durch die Eigenschaft `rdfs:type cims:ClassCategory` ausgezeichnet. Zudem müssen Packages ein `rdfs:label` haben und können optional noch ein `rdfs:comment` haben. Label und Kommentar sind auf allen Ressourcentypen gleich definiert und werden daher im Folgenden nicht mehr explizit erwähnt.

Code 2.5 zeigt ein Beispiel für die Definition eines Packages, welches die Lehrveranstaltungen einer Hochschule gruppiert.

```
ex:Package_Lehrveranstaltungen
    a cims:ClassCategory ;
    rdfs:label "Lehrveranstaltungen"@en ;
    rdfs:comment "Ein Package fuer Lehrveranstaltungen."@de .
```

Code 2.5: Package Beispiel

Während Packages in den CGMES-Profilen mit der Version 2.4.15 noch genutzt wurden, werden Klassen in Profilen der Version 3.0 meistens in einem einzigen Package zusammengetragen.

CIM-Klassen

CIM-Klassen werden durch die Eigenschaft `rdfs:type cims:Class` ausgezeichnet. Mittels `cims:belongsToCategory` können Klassen einem Package zugeordnet werden und mittels `rdfs:subClassOf` kann eine erbende Relation zu einer anderen Klasse dargestellt werden. Zudem können der Klasse mit `cims:stereotype` verschiedene Typ-Informationen gegeben werden. Stereotypen können beliebig gewählt werden, und es können einer Klasse mehrere Stereotypen zugewiesen werden. Demnach ist es das einzige Prädikat, welches auf einer Ressource nicht einzigartig ist.

Die für diese Arbeit relevanten Klassen-Stereotypen sind:

- `http://iec.ch/TC57/NonStandard/UML#concrete`: Kennzeichnet eine Klasse als instanziierbar. Eine Abwesenheit dieses Stereotyps markiert eine Klasse demnach als abstrakt.
- `http://iec.ch/TC57/NonStandard/UML#enumeration`: Kennzeichnet eine Klasse als eine Enum Klasse, die verschiedene Enum Entries bereit stellt.
- `"Primitiveünd "CIMDatatype"`: Kennzeichnen eine Klasse als einen selbst definierten Datentyp, der von Attributen anderer Klassen genutzt werden kann.

Code 2.6 zeigt ein Beispiel für die Definition einer Klasse, die ein Modul Compilerbau repräsentiert.

```
ex:Compilerbau
    a rdfs:Class ;
    rdfs:label "Compilerbau"@de ;
    rdfs:subClassOf ex:Modul ;
    cims:belongsToCategory ex:Package_Lehrveranstaltungen ;
    cims:stereotype <http://iec.ch/TC57/NonStandard/UML\#concrete> ;
```

Code 2.6: Klasse Beispiel

CIM-Attribute

CIM-Attribute werden nicht direkt als Teil einer Klasse definiert, sondern liegen als eigenständige Ressource mit `rdf:type rdf:Property` vor. Sie sind durch `cims:stereotype <http://iec.ch/TC57/NonStandard/UML#attribute>` gekennzeichnet und werden einer Klasse mit dem `rdfs:domain` Prädikat zugewiesen. Die Zugehörigkeit zu einer Klasse wird außerdem meist in der IRI angegeben. Diese setzt sich zusammen aus einem Präfix, dem Klassennamen und dem Attribut-Namen, wobei Klasse und Attribut durch einen Punkt getrennt werden. Hierbei ist darauf zu achten, dass der Präfix des Attributes nicht gleich dem der Klasse sein muss.

Der Datentyp eines Attributes kann entweder mit `rdfs:range` oder `cims:dataType` angegeben werden. `rdfs:range` wird verwendet, wenn der Typ des Attributs eine Enum Klasse ist, wobei der Wert die IRI eines Enum Entries sein muss. `cims:dataType` wird hingegen bei allen anderen literalen Datentypen verwendet. Zudem können einem Attribut mit `cims:isFixed` und `cims:isDefault` fixe bzw. Default-Werte zugewiesen werden.

Mit `cims:multiplicity` wird angegeben, ob ein Attribut ein Pflichtfeld oder optional ist. Üblicherweise ist die Multiplizität aufgebaut als `M:<Minimum>..<Maximum>`.

`M:0..1` bedeutet hierbei, dass das Attribut keinmal oder einmal vorkommen kann, während M:1..1 bedeutet, dass das Attribut exakt einmal vorkommen muss.

Code 2.7 zeigt ein Beispiel für die Definition eines Attributs Matrikelnummer für die Klasse Student.

```
ex:Student.Matrikelnummer
    a rdf:Property ;
    rdfs:label "Matrikelnummer"@de ;
    rdfs:domain ex:Student ;
    cims:dataType xsd:integer ;
    cims:stereotype <http://iec.ch/TC57/NonStandard/UML#attribute> ;
    cims:multiplicity cims:M:1..1 ;
    cims:isDefault "0000000"^^xsd:integer .
```

Code 2.7: Attribut Beispiel

Datentypen

Datentypen für Attribute werden in Form einer IRI angeben, wobei zwischen drei verschiedenen Arten von Datentypen unterschieden wird: XSD-Datentypen, primitive Datentypen und CIM-Datentypen.

XSD-Datentypen sind vordefinierte Datentypen, die im Rahmen der XML Schema Definition (XSD) definiert wurden. Diese XSD-Datentypen umfassen die meisten primitiven Datentypen, die man auch von Programmiersprachen kennt, wie z.B. `xsd:string` oder `xsd:boolean`.

Primitive Datentypen sind Datentypen die im Graphen selbst definiert sind und den Stereotyp „Primitive“ haben. Sie dienen meistens als einfache Wrapper für die XSD-Datentypen und beinhalten ähnliche Datentypen wie `cim:String` doer `cim:Boolean`.

CIM-Datentypen werden ebenfalls im Graphen selbst definiert, bilden jedoch komplexere Datentypen ab. Sie haben immer jeweils drei Attribute: `value`, `unit` und `multiplier`, welche den zugrundeliegenden Datentyp, die Einheit und den Multiplikator des Datentyps angeben. Ein Beispiel für einen CIM-Datentyp wäre `cim:Voltage`, welcher das value `cim:double`, die unit `V` und den multiplier `k` hat.

CIM-Assoziationen

CIM-Assoziationen sind immer als Paare definiert, sowohl in der Hin- als auch in der Rückrichtung, und können genutzt werden, um Beziehungen zwischen zwei Klassen zu modellieren. Code 2.8 definiert z.B. die Beziehung zwischen den Klassen Professor und Modul.

```
ex:Professor.Modul
    rdf:type rdf:Property ;
    rdfs:label "Module"@de ;
    rdfs:domain ex:Professor ;
    rdfs:range ex:Modul ;
    cims:AssociationUsed "Yes" ;
    cims:inverseRoleName ex:Modul.Professor ;
    cims:multiplicity cims:M:0..n .

ex:Modul.Professor
    rdf:type rdf:Property ;
    rdfs:label "Professor"@de ;
    rdfs:domain ex:Modul ;
    rdfs:range ex:Professor ;
    cims:AssociationUsed "Yes" ;
    cims:inverseRoleName ex:Professor.Modul ;
    cims:multiplicity cims:M:1..1 .
```

Code 2.8: Assoziation Beispiel

Wie Attribute sind Assoziationen vom Typ `rdf:type rdf:Property` und werden mittels `rdfs:domain` einer Klasse zugewiesen. Zusätzlich haben sie eine `rdfs:range`, welche die Target-Klasse definiert. Die IRI setzt sich aus dieser Domain und Target-Klasse zusammen, getrennt durch einen Punkt. Auch hier ist der Präfix nicht zwingend der Gleiche wie der der Domain.

Ebenso ähnlich zu den Attributen ist die Definition der Multiplizität. Diese nimmt bei Assoziationen jedoch auch häufiger andere Werte als `M:0..1` und `M:1..1` an. So werden z.B. nach oben offene Multiplizitäten durch ein n angegeben. Diese Schreibweisen können zudem auch abgekürzt werden; falls die minimale Multiplizität gleich 0 ist wird sie oft weggelassen und als `M::<Maximum>` geschrieben. In Code 2.8 kann man auch sehen, dass die Multiplizität in beide Richtungen zudem verschieden sein kann. Während ein Professor mehrere Module anbieten kann, ist ein Modul immer nur einem Professor zugeordnet.

Da die Assoziationen jeweils paarweise definiert sind verweisen sie mit `cims:inverseRoleName` auf die andere Assoziation im Paar. Außerdem kann mit

`cims:AssociationUsed` angegeben werden, ob eine Assoziation tatsächlich instanziiert werden soll, oder nur als Referenz definiert ist.

CIM-Enum-Entries

Enum-Entries bilden die verschiedenen Werte ab, die eine Enum Klasse haben kann. Sie werden der Klasse mittels des `rdf:type` Prädikats zugeordnet, wobei das Objekt die IRI der Enum Klasse ist. Ähnlich wie auch bei Attributen und Assoziationen setzt sich die IRI aus einem Präfix, dem Label der Klasse, einem Punkt und dem eigenen Label zusammen.

```
ex:Klausurergebnis.BESTANDEN
    a ex:Klausurergebnis
    rdfs:label "BESTANDEN"@de
```

Code 2.9: Enum Entry Beispiel

2.2.4 CIM-Modelle und Instanzdaten

Während in relationalen Datenbanken die Struktur der Daten immer direkt durch die Tabelle definiert wird, in der auch die Instanzdaten gespeichert sind, sind Schema und Instanzdaten in CIM nicht direkt aneinander gebunden. Daher wird in den meisten Fällen ein neuer Graph für die Instanzdaten eines Schemas angelegt. Die IRI der Klasse wird hierbei mit `rdf:type` angegeben und alle Properties, die die Klasse hat, werden mit der IRI des Properties als Prädikat definiert. Dabei ist zu beachten, dass auch alle geerbten verpflichtenden Properties, die nicht auf der Klasse selbst definiert wurden, instanziiert werden müssen. Da die IRI von Properties immer die Domain enthält, lässt sich hier auch weiterhin nachvollziehen, wo ein geerbtes Property herkommt.

Bei der Instanziierung von Attributen ist zu beachten, dass Literale immer nur mit den primitiven XSD-Datentypen typisiert werden können. Hat ein Attribut daher einen selbst definierten Datentyp wie beispielsweise `ex:Voltage`, dann wird dieser in den Instanzen auf den XSD-Datentyp `xsd:integer` abgeleitet.

Code 2.10 zeigt, wie eine Klasse „Student“ instanziiert werden könnte. Die IRI wird dabei auf eine zufällig generierte UUID gesetzt, um Kollisionen zu vermeiden.

```
ex:4a8e2df0-a47f-43ed-a673-b00b7bcd2ed7
    a ex:Student .
    ex:Person.Name "Max Mustermann" .
    ex:Student.MatrikelNummer 1324567 .
```

Code 2.10: Instanzen Beispiel

2.3 Shapes Constraint Language

Die Shapes Constraint Language (SHACL) ist eine Sprache, die genutzt werden kann, um RDF-Graphen gegen spezifische Anforderungen zu validieren. Zwar werden bestimmte Eigenschaften wie die Kardinalität und der Datentyp bereits durch `cims:multiplicity` und `cims:dataType` beschrieben, es fehlt jedoch die Möglichkeit, diese Eigenschaften zu validieren oder komplexere Constraints zu definieren. [19]

Mit SHACL lassen sich ähnlich zu den Constraints in SQL sehr viel detailliertere Anforderungen an Ressourcen stellen, wie zum Beispiel ein Integerwert, der zwischen 0 und 255 liegen muss. Für jede Eigenschaft, die validiert werden soll, wird ein sogenannter SHACL-Shape erstellt. [19]

Code 2.11 zeigt beispielhaft einen Shape, der sicherstellt, dass eine Matrikelnummer einen Integer-Wert zwischen 1000000 und 9999999 hat.

```
ex:MatrikelnummerWertebereichShape
    a sh:PropertyShape ;
    sh:path ex:Matrikelnummer ;
    sh:description "Validiert den Wertebereich der Matrikelnummer" ;
    sh:minInclusive 1000000 ;
    sh:maxInclusive 9999999 ;
    sh:message "Die Matrikelnummer muss zwischen 1000000 und 9999999 liegen" .
```

Code 2.11: SHACL-Shape Beispiel

Der genaue Aufbau der SHACL-Shapes und die Bedeutung der hier genutzten Prädikate ist für diese Arbeit nicht relevant, lediglich die Tatsache, dass SHACL für die Validierung von Instanzdaten genutzt werden kann.

Wenn man die Instanzdaten gegen die SHACL-Regeln eines Datensatzes validiert, wird ein sogenannter Validation Report erstellt. Dieser Report gibt für jeden erkannten Fehler den relevanten SHACL-Shape sowie die betroffene Ressource an. Anhand dieses Reports kann der Anwender überprüfen, ob die Instanzdaten für

ein Modell valide sind. Falls die Daten nicht valide sind, kann der Report zudem genutzt werden, um fehlerhafte Daten zu identifizieren und zu korrigieren. [20]

2.4 String-Ähnlichkeit

Die Ähnlichkeit zwischen zwei Zeichenketten zu bestimmen ist Kernbestandteil von vielen verschiedenen Disziplinen, von Suchmaschinen über Spam Erkennung bis hin zum Analysieren von DNA. Dafür stehen eine Vielzahl von verschiedenen Algorithmen zur Verfügung. Generell lassen sich diese Verfahren in drei verschiedene Kategorien unterteilen: Edit-basierte, Sequence-basierte und Token-basierte Algorithmen [21]. Dieser Abschnitt gibt einen Überblick über diese Kategorien und einige ihrer Algorithmen.

2.4.1 Editbasiert

Editbasierte Algorithmen vergleichen zwei Strings anhand der Anzahl an Änderungen, oder auch Edits, die durchgeführt werden müssen, um einen String in den anderen zu transformieren. Die einfachste Implementierung hierfür bietet die Hamming-Distanz. Dieser Algorithmus iteriert über beide Strings und ermittelt die Anzahl der Positionen, an denen sie nicht übereinstimmen. Die Anzahl an Positionen, an denen die beiden Strings voneinander abweichen wird als deren Distanz bezeichnet, wobei eine kleinerer Wert eine höhere Ähnlichkeit bedeutet. Die beiden Wörter „Boot“ und „Beet“ hätten beispielsweise eine Hamming-Distanz von zwei. [22]

Ein grundlegendes Problem der Hamming-Distanz fällt sofort auf: es können nur Distanzen für Strings gleicher Länge berechnet werden. Die Levenshtein-Distanz löst dieses Problem, indem der Algorithmus nicht nur das Verändern eines Zeichens als mögliche Änderung betrachtet, sondern auch das Löschen oder Hinzufügen. So hätten „Schild“ und „Bild“ eine Levenshtein-Distanz von drei, mit einem geänderten und zwei gelöschten Buchstaben. [23]

Sowohl die Hamming-Distanz als auch die Levenshtein-Distanz begünstigen durch das Bestimmen von nicht übereinstimmeden Zeichen kurze Zeichenketten, da mit zunehmender Länge der Zeichenkette die Anzahl möglicher Änderungen steigt. Möchte man die errechneten Ergebnisse mit denen Distanzen anderer Inputs oder anderer Algorithmen vergleichen, müssen die Werte vorher normalisiert werden. Eine einfache Lösung ist es hierbei, die Werte durch die Länge der Inputs zu teilen, wodurch sich ein Wert zwischen 0 und 1 ergibt.

Eine weitere Verbesserung gegenüber der Levenshtein-Distanz bietet die Jaro-Ähnlichkeit. Sie betrachtet zwei gleiche Zeichen als übereinstimmend, auch wenn sie eine gewissen Abstand im String zueinander haben. Der zulässige Abstand ist dabei definiert als die Hälfte der Länge des längeren Strings, abgerundet. Durch diese Erweiterung lassen sich versehentliche Vertauschungen von Buchstaben erkennen, die bei den anderen Algorithmen als zwei geänderte Zeichen erkannt werden. [24]

Der Jaro-Winkler-Algorithmus baut auf diesem Konzept weiter auf, indem er Übereinstimmungen am Wortanfang stärker gewichtet, da Tippfehler dort seltener auftreten [25]. Sowohl Jaro als auch Jaro-Winkler errechnen keine Distanz sondern einen Ähnlichkeitswert, wobei ein größerer Wert eine höhere Ähnlichkeit bedeutet. Im Gegensatz zu Distanzen sind diese bereits normiert. Da Ähnlichkeit und Distanz jedoch entgegengesetzte Metriken sind, müssen die Werte vor einem Vergleich gegebenenfalls noch umgerechnet werden.

2.4.2 Sequenzbasiert

Sequenzbasierte Algorithmen versuchen nicht Änderungen an einzelnen Zeichen zu identifizieren, sondern betrachten stattdessen die gesamte Zeichenkette. Der Algorithmus „Longest Common Substring“ analysiert beide Strings mithilfe einer Häufigkeitsmatrix, um die längste kontinuierliche, gemeinsame Zeichenkette zu finden. Der verwandte Algorithmus „Longest Common Subsequence“ arbeitet ähnlich, setzt dabei allerdings nicht voraus, dass die gemeinsame Subsequence kontinuierlich ist und erlaubt Lücken. Betrachten wir zur Erläuterung die folgenden zwei Sätze:

> The quick brown fox jumps over the lazy dog
>
> The fox jumps over the dog

Der Longest Common Substring der beiden Sätze ist hier nur „fox jumps over the“, während die Longest Common Subsequence „The fox jumps over the dog“ umfasst. Das „The“ am Anfang und „dog“ am Ende können in die Longest Common Subsequence aufgenommen werden, da „quick brown“ und „lazy“ übersprungen werden können, auch wenn sie nicht übereinstimmen.

Ähnlich zu Distanzwerten müssen die Ergebnisse dieser beiden Algorithmen ebenfalls normalisiert werden, bevor sie mit den Ergebnissen anderer Algorithmen verglichen werden können. Hierfür können ebenfalls die Längen der Sequenzen durch die Länge des längeren Strings geteilt werden.

2.4.3 Tokenbasiert

Tokenbasierte Algorithmen unterteilen die zu analysierenden Strings vor dem Vergleich in kleinere Abschnitte, sogenannte Tokens. Eine häufige Methode, die Strings zu unterteilen, ist das „Bag of Words“ Verfahren. Diese Methode wird vorwiegend auf Sätzen angewendet und unterteilt diese in ihre einzelnen Wörter, kann aber auch genutzt werden, um Bezeichner in CamelCase-Notation in einzelne Wörter aufzuteilen.

Eine weitere Möglichkeit, Tokens zu generieren, sind N-Grams. Dabei wird der String in Tokens einer festen Länge unterteilt. Allerdings werden die N-Grams mithilfe eines fortlaufenden Fensters erstellt und überlappen sich dadurch. Die 2-Grams, auch Bigrams genannt, von „Hallo“ wären demnach {Ha, al, ll, lo}. Welche Methode der Tokengenerierung verwendet wird, hängt vom Anwendungsfall ab, und die Algorithmen können in den meisten Fällen mit jeder Methode umgehen.

Die einfachste Methode, die Ähnlichkeit von zwei Mengen von Tokens zu ermitteln. ist der Jaccard Index. Dabei wird die Größe des Schnitts der beiden Sets durch die Größe der Vereinigung geteilt. Es wird also die Prozentzahl der Tokens berechnet, die in beiden Strings vorkommen. [26]

Etwas abstrakter ist die „Kosinus-Ähnlichkeit“. Hier werden die Tokens und ihre Häufigkeit als Vektoren im n-dimensionalen Raum betrachtet und es wird der Winkel zwischen beiden errechnet. Der Vorteil von der Kosinus-Ähnlichkeit gegenüber Jaccard ist, dass die Häufigkeitsinformationen erhalten bleiben. Während Jaccard die Tokens als Set betrachtet, fließen bei der Kosinus-Ähnlichkeit die Häufigkeiten in das Skalarprodukt mit ein. [27]

2.5 Technologien

Die folgenden Abschnitte geben einen Überblick über die Technologien, die in dieser Arbeit verwendet wurden. Die Auswahl der Technologien erfolgte dabei anhand der bereits verwendeten Technologien im RDF-Architect und der Eignung für die jeweiligen Aufgaben. Zudem müssen alle verwendeten Bibliotheken mit der Apache 2.0 Lizenz kompatibel sein, da der RDF-Architect in Zukunft als Open-Source Software unter dieser Lizenz veröffentlicht werden soll.

2.5.1 Backend

Spring Boot

Das Java Framework Spring wird von VMWare entwickelt, mit dem Ziel, die Entwicklung von Enterprise Applikationen in Java zu vereinfachen. Springs Kernfunktionalität besteht in dem automatischen Instanziieren und Managen von Dependencies zwischen Komponenten anhand des sogenannten „Inversion of Control“ Prinzips. Dadurch wird Boilerplate-Code reduziert und die Applikation ist einfacher zu testen. [28]

Das ebenfalls von VMWare entwickelte Spring Boot Framework erweitert diese Funktionalitäten zusätzlich, indem es dem Nutzer noch mehr der Konfigurationsarbeit abnimmt. Statt Konfiguration durch XML Dateien erkennt Spring Boot automatisch Dependencies und entscheidet eigenständig, wie diese konfiguriert werden müssen. Zudem bietet Spring Boot einen eigenen, mitgelieferten Server und weitere nützliche Bibliotheken. [29]

Für die Entwicklung von Webanwendungen stellt Spring einige Annotationen bereit, wie etwa `@RestController` oder `@RequestMapping`. Diese ermöglichen die einfache, deklarative Konfiguration von API-Endpunkten. Das Framework übernimmt dabei das Mapping von JSON-Objekten, Error-Handling und Erstellen von HTTP-Responses. Diese Eigenschaften machen Spring Boot besonders geeignet für die Entwicklung der RESTful API des RDF-Architect. [30]

Apache Jena RDF API

Apache Jena ist ein Open-Source Framework der Apache Foundation und bietet umfassende Schnittstellen für die Arbeit mit RDF-Graphen. Das Framework erlaubt unter anderem das Serialisieren und Deserialisieren von Graphen in alle gängigen RDF-Formate, das Bearbeiten der Graphen mittels eines Query-Builders sowie die Validierung von Daten durch SHACL-Regeln. [31]

Für die Implementierung der Schemamigration ist insbesondere die Funktionalität der Parameterized SPARQL Strings zentral. Diese ermöglichen es, SPARQL-Update-Templates mit Platzhaltern zu definieren, die zur Laufzeit anhand der gefundenen Änderungen mit Werten befüllt werden. Diese werden anschließend zu einem vollständigen Migrationsskript kombiniert. Diese Vorgehensweise erlaubt eine flexible Generierung des Migrationsskript, auch in Fälle, in denen viele Änderungen vorliegen.

Apache Commons Text

Apache Commons ist, wie auch Apache Jena, ein Projekt der Apache Foundation, mit dem Ziel, wiederverwendbare Java-Komponenten zu entwickeln. Apache Commons Text ist eine im Rahmen dieses Projektes entwickelte Bibliothek mit verschiedenen Algorithmen, die die Arbeit mit Strings ermöglichen [32]. Das Similarity Package enthält Algorithmen spezifisch für die Berechnung von Ähnlichkeitswerten zwischen Strings.

Die Einbindung der Stringvergleichsalgorithmen bietet dabei mehrere Vorteile gegenüber einer eigenen Implementierung: Die Algorithmen sind bereits umfassend getestet, optimiert und durch die breite Nutzung in der Community auf Korrektheit getestet. Zudem spart die Nutzung bestehender Lösungen Zeit, die stattdessen in die Funktionalitäten der Schemamigration investiert werden kann.

Die Wahl der Apache Commons Text Library erfolgte aufgrund ihrer Apache 2.0-Lizenz, die mit den Lizenzanforderungen des RDF-Architect kompatibel ist, sowie der Reputation der Apache Foundation als vertrauenswürdige Quelle. Die Bibliothek stellt mehrere Algorithmen zur Verfügung, was einen Vergleich untereinander ermöglicht, und ist verglichen mit anderen Bibliotheken wie SimMetrics relativ leichtgewichtig.

Das Similarity Package stellt folgende Algorithmen zur Verfügung:

- Cosine Distance
- Cosine Similarity
- Jaccard Similarity
- Fuzzy Score
- Hamming Distance
- Jaro-Winkler Distance
- Jaro-Winkler Similarity
- Levenshtein Distance
- Longest Common Subsequence Distance

Die Algorithmen arbeiten jeweils wie in Abschnitt 2.4 beschrieben. Eine Ausnahme stellt der Fuzzy Score dar, bei dem es sich nicht um einen gängigen Algorithmus handelt. Der Fuzzy Score ist ähnlich zu editbasierten Algorithmen und gibt Punkte für jedes identische Zeichen der beiden Strings, wobei extra Punkte für mehrere aufeinanderfolgende identische Zeichen gegeben werden.

Eine weitere Besonderheit stellen die tokenbasierten Algorithmen dar. Die Jaccard Similarity und Cosine Distance nehmen jeweils zwei Strings als Input und zerlegen diese eigenständig in Tokens. Jaccard erstellt Tokens für die einzelnen Zeichen, während Cosine Distance Wörter als Tokens verwendet. Die Cosine Similarity hingegen erwartet als Input bereits zwei Listen von Tokens, weswegen die Tokengenerierung vor dem Aufruf durchgeführt werden muss. Dies erlaubt jedoch auch eine flexiblere Generierung der Tokens und beispielsweise die Nutzung von Bigrams.

Project Lombok

Project Lombok ist eine Java Bibliothek zur Reduzierung von Boilerplate-Code durch die Generierung von Standard-Methoden. Die Bibliothek stellt verschiedene Annotationen zur Verfügung, die genutzt werden können, um Konstruktoren, Getter und Setter oder Equals-Methoden von Klassen automatisch zu generieren. Dies spart Zeit in der Entwicklung und verhindert Fehler durch vergessenes Anpassen von Methoden beim Hinzufügen von neuen Feldern. [33], [34]

Da für die Umsetzung der Schemamigration viele neue Datenobjekte angelegt werden müssen, beschleunigt die Nutzung von Lombok Annotationen die Entwicklung und erlaubt das einfache Hinzufügen von neuen Feldern auf diesen Datenobjekten, ohne dass Methoden angepasst werden müssen.

2.5.2 Frontend

Das Frontend des RDF-Architects nutzt moderne Webtechnologien für eine reaktive und performante Benutzeroberfläche. Die gewählten Technologien ermöglichen eine modulare und komponentenbasierte Entwicklung.

Svelte 5

Svelte ist ein modernes JavaScript-Framework für die Erstellung von reaktiver UI. Im Gegensatz zu anderen Frameworks wie Vue oder React arbeitet Svelte nicht mit einem Virtual DOM, sondern kompiliert die Svelte-Dateien bereits während des Build-Prozesses in optimiertes JavaScript, welches das DOM direkt manipuliert. Dadurch können die Größe der übertragenen Dateien minimiert und Performance verbessert werden. [35]

Den Kern von Sveltes Reaktivität bilden die reaktiven Variablen, die bei einer Änderung automatisch ein Rerendern auslösen. Diese können mittels der `$state` Rune deklariert werden. Auch Variablen, die mittels `$props` zwischen Eltern-Kind-Komponenten geteilt werden, sind automatisch reaktiv. [36]

Obwohl Svelte vergleichsweise jung ist, wird es bereits von vielen großen Unternehmen genutzt und hat eine aktive Community.

Svelte Kit

Svelte Kit erweitert das reine UI-Framework von Svelte um die nötigen Funktionalitäten zur Erstellung von kompletten Webanwendungen. Es bietet unter anderem ein eigenes Routing-System, Server-Side Rendering, API-Endpunkte und file-based Routing. Zusammen mit Svelte kann Svelte Kit genutzt werden, um moderne und reaktive Single Page Applications zu erstellen. [37]

Tailwind

Tailwind CSS ist ein Utility-First und Open-Source CSS Framework. Im Gegensatz zu anderen CSS Frameworks wie Bootstrap stellt es keine vorgefertigten Komponenten wie Buttons oder Tabellen zur Verfügung, sondern bietet eine Vielzahl von modularen CSS-Klassen, die jeweils eine einzelne Eigenschaften definieren. Durch das flexible Kombinieren dieser Klassen können so auch komplexe Designs erstellt werden, ohne dafür eigene CSS-Dateien schreiben zu müssen.

Tailwind kompiliert diese Utility Klassen in klassisches CSS, wobei ungenutzte oder duplizierte Styles entfernt werden. Dadurch bleiben die die übertragenen CSS-Dateien klein und performant. [38], [39]

2.5.3 Architektur

Hexagonales Backend

Das Backend des RDF-Architects folgt dem Prinzip der hexagonalen Architektur, auch bekannt als Ports- und Adapter-Architektur. Dabei wird die Applikation in verschiedene Schichten und Komponenten aufgeteilt. Den Kern der Applikation bildet die Domänenschicht, die die Geschäftslogik der Applikation enthält. Außerhalb dieses Kerns befinden sich die Datenbank- und die API-Schicht, welche über definierte Schnittstellen, den sogenannten Ports, mit der Domänenschicht kommunizieren. Durch diese Trennung bleibt die Geschäftslogik unabhängig von den konkreten Implementierungen der Datenbank oder API, und erlaubt sogar, diese beliebig auszutauschen. [40]

REST

Das Konzept des Representational State Transfer (REST) beschreibt einen Architekturstil für API-Endpunkte. REST-Schnittstellen nutzen die etablierten HTTP-Methoden wie GET, POST, PUT und DELETE und wendet diese auf Ressourcen-orientierte URLs an.

Die Kommunikation mit REST-APIs erfolgt stateless, die Anfragen enthalten also alle nötigen Informationen, um vom Server verarbeitet zu werden. Informationen können entweder über URL-Parameter, als JSON-Objekte im Body oder über den HTTP-Header mitgeliefert werden.

Im RDF-Architect wird eine REST-Schnittstelle für die Kommunikation zwischen dem Svelte-Frontend und dem Spring Boot Backend genutzt.

3 RDF-Architect

Der RDF-Architect ist eine von SOPTIM entwickelte Webanwendung zum Erstellen und Bearbeiten von CIM-Modellen. Bisher erfolgt die Pflege der Modelle bei der SOPTIM und ihren Kunden mit dem von Sparx Systems entwickelten Enterprise Architect und erweiternden Plugins. Aufgrund mehrerer Fehler in diesen Plugins ist die Arbeit mit dem Enterprise Architect jedoch umständlich und führt häufig zu invaliden Daten.

Um diese Probleme zu beheben und der Branche eine bessere Lösung bereitzustellen, hat sich die SOPTIM entschieden, eine eigene Software zu entwickeln und auf Open Source Basis zu veröffentlichen. Da die Anwendung der Entwicklung von Modellen dient, ein Aufgabe des Softwarearchitekten, und sich auf RDF-basierte Graphen fokussiert, wurde sie „RDF-Architect“ genannt.

Dieses Kapitel soll einen Überblick über die bisherigen Funktionalitäten des RDF-Architect geben. Dafür werden in Abschnitt 3.1 die Kernfunktionalitäten der Anwendung beschrieben, wobei besonders auf den Schemavergleich eingegangen wird, der als Grundlage für die Migrationsfunktionalität dient. In Abschnitt 3.2 und Abschnitt 3.3 wird anschließend die Architektur der Anwendung erläutert und beispielhaft auf die Implementierung des Backends und Frontends eingegangen.

3.1 Bestehende Funktionalitäten

Der RDF-Architect bietet alle grundlegenden Funktionalitäten eines herkömmlichen UML-Editors, wie das Erstellen, Bearbeiten und Löschen von Klassen, Attributen und Assoziationen. Dabei ist die Anwendung allerdings stark auf den Kontext von CIM/CGMES-konformen Schemata ausgelegt und bietet einige spezielle Funktionalitäten, die in herkömmlichen UML-Editoren nicht vorhanden sind.

Abbildung 3.1 zeigt die Hauptseite des RDF-Architects, die die Klassendiagramme darstellt sowie Möglichkeiten zur Bearbeitung bietet. Eine größere Version der Übersicht ist im Anhang unter Abbildung A.1 zu finden.

Auf der linken Seite befindet sich die Navigationsleiste, über die verschiedene Diagramme ausgewählt werden können. Die Navigation ist ähnlich zu der Navigation in einem Datei-Explorer aufgebaut, wobei ein Datensatz mehrere Graphen und ein Graph mehrere Packages enthält. Ein Diagramm stellt immer ein CIM-Package

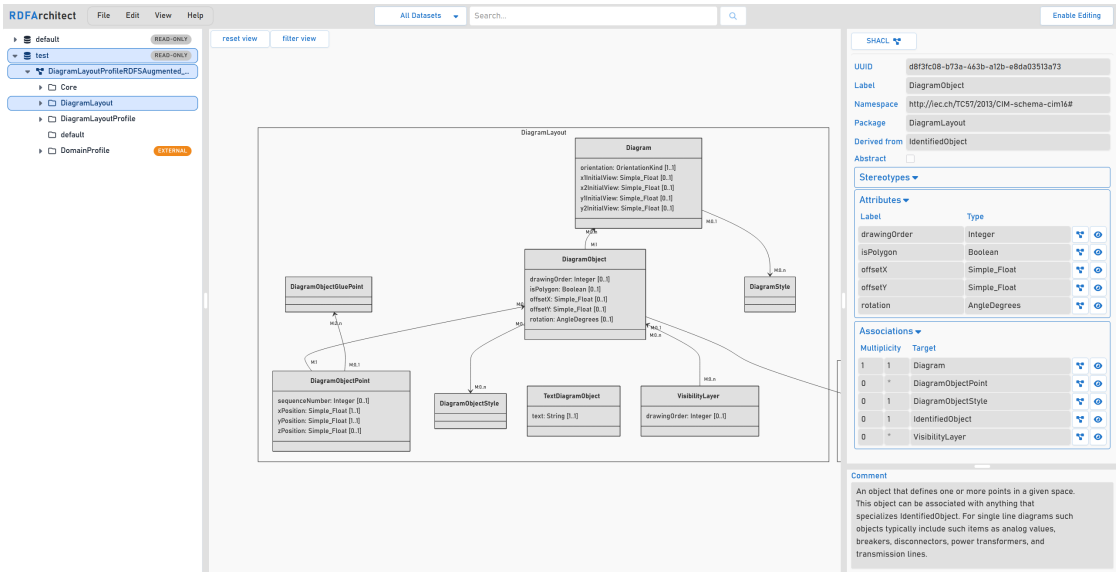


Abbildung 3.1: RDF-Architect Übersicht

dar und zeigt alle darin enthaltenen Klassen, sowie potentielle Verbindungen zu Klassen aus anderen Packages, als UML-Diagramm an.

Wählt man eine der Klassen im Diagramm aus, so öffnet sich der auf der rechten Seite angezeigte „Class Editor“. Dieser ermöglicht das Bearbeiten aller Eigenschaften der Klasse selbst und deren Properties.

Neben dem Editor als Kernfunktionalität bietet der RDF-Architect noch einige weitere nützliche Features. Die Suchleiste am oberen Rand ermöglicht beispielsweise das schnelle Finden von Ressourcen über verschiedenen Datensätze hinweg und über ein Popup-Fenster lässt sich ein Change-Log eines Graphen verwalten.

3.1.1 Schemavergleich

Von besonderem Interesse für diese Arbeit ist der bereits implementierte Schemavergleich (s. Abbildung 3.2 bzw. Abbildung A.2 im Anhang). Der Schemavergleich ermöglicht es, zwei Versionen eines RDF-Schemas oder zwei verschiedene RDF-Schemata miteinander zu vergleichen. Die Ansicht enthält auf der linken Seite eine ähnliche Navigationsleiste wie die Hauptseite, über die die Unterschiede in den verschiedenen Packages betrachtet werden können.

Die Änderungen zwischen den Schemata werden in der Mitte in Tabellenform dargestellt, wobei immer eine Tabelle pro geänderter Ressource angezeigt wird. Die Zeilen der Tabelle enthalten jeweils die IRI des geänderten Prädikats sowie den

alten und neuen Wert des Objekts. Additionen werden dabei grün, Löschungen rot und Änderungen grau hervorgehoben. Diese Farbkodierung wird auch auf die Packages in der Navigationsleiste angewendet, falls ein gesamten Package gelöscht oder hinzugefügt wurde.

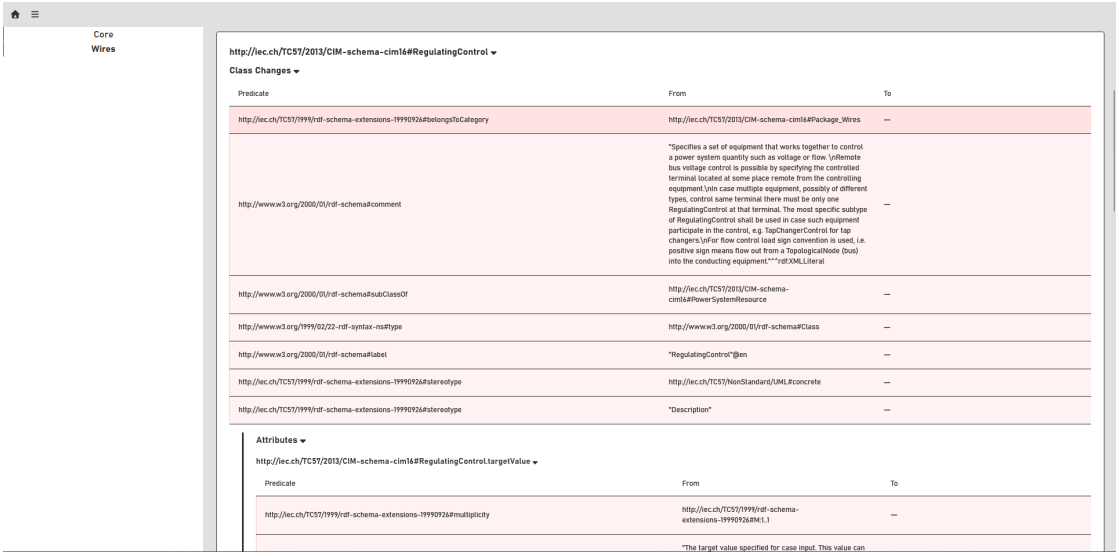


Abbildung 3.2: Schemavergleich

Änderungen an Properties einer Klasse werden in eigenen Tabellen dargestellt, die in den Sektionen der zugehörigen Klasse eingebettet sind. Dabei lassen sich die Tabellen und Sektionen auf- und zuklappen, um sich auf relevante Änderungen konzentrieren zu können.

Trotz dieser Maßnahmen, um die Oberfläche übersichtlicher zu gestalten, ist der Schemavergleich bei größeren Änderungen immer noch schwer zu überblicken. Besonders wenn Klassen mit vielen Attributen gelöscht oder hinzugefügt werden, führt dies zu sehr vielen geänderten Tripeln, obwohl sich die semantische Änderung nur auf das Löschen oder Hinzufügen einer einzigen Klasse beschränkt. Ebenso lassen sich Umbenennungen nicht direkt erkennen, da Ressourcen in RDF anhand ihrer IRI identifiziert werden und eine Umbenennung somit als Löschen der alten und Hinzufügen einer neuen Ressource interpretiert wird.

Es braucht daher noch viel manuelle Analyse, um anhand der dargestellten Tripel-Änderungen die relevanten semantischen Änderungen zu erkennen. Dennoch bildet diese Funktionalität eine gute Grundlage, auf der die Implementierung der Schemamigration aufbauen kann, sowohl durch die bestehenden UI-Konzepte als auch bereits angelegte Modelle und Methoden im Backend.

3.2 Backend

Das Backend des RDF-Architects ist in Java geschrieben und nutzt das Spring Boot Framework. Die Anwendung folgt dem Prinzip einer hexagonalen Architektur, bei der die Geschäftslogik von der Datenbank und den REST-Controllern getrennt ist. Die Kommunikation mit dem Frontend erfolgt über eine klassische REST-API, bei der JSON als Datenformat genutzt wird. Das Backend nutzt eine auf Apache Jena aufbauende In-Memory-Datenbank, die über die Jena Query API abgefragt wird.

Neben Spring Boot als Framework kommen verschiedene weitere Bibliotheken zum Einsatz, darunter Lombok zur Reduzierung von Boilerplate-Code, Jackson für die JSON-Serialisierung und OpenAPI für die automatische Generierung der API-Dokumentation. Wie diese Bibliotheken genutzt werden, um die verschiedenen Komponenten des Backends zu implementieren, wird im Folgenden beschrieben.

3.2.1 Nutzung der hexagonalen Architektur

Das Backend des RDF-Architects folgt einer hexagonalen Architektur, bei der die Geschäftslogik von den REST-Controllern und der Datenbank durch Ports und Adapter getrennt sind. Diese Ports werden in Java durch die Nutzung von Interfaces realisiert, die von den Services implementiert werden. Pro API-Endpunkt wird ein Interface, im RDF-Architect als Use Case bezeichnet, erstellt, welches von den Endpunkten aufgerufen wird. Diese Interfaces werden von den Services implementiert, wobei ein Service auch mehrere Interfaces implementieren kann. Spring Boot übernimmt dabei automatisch die Initialisierung der Services und die Injektion in die Controller.

Die Anbindung an die Datenbank folgt dem gleichen Prinzip. Services rufen Daten aus der Datenbank über ein Interface ab, welches von einer Implementierungsklasse realisiert wird. Dadurch können verschiedene Implementierungen der Interfaces genutzt werden, ohne dass die Services angepasst werden müssen.

3.2.2 Projektstruktur

Die Implementierung des Backends des RDF-Architects ist in mehrere Packages aufgeteilt, die jeweils unterschiedliche Verantwortlichkeiten haben. Die wichtigsten Packages für diese Arbeit sind:

- api: Enthält Controller-Klassen, die die Endpunkte definieren.
- models: Definiert die in den Services verwendeten Datenmodelle.

- services: Beinhaltet die Geschäftslogik der Anwendung.

Weitere Packages für die Implementierung der Datenbank oder für Exception-Handling existieren ebenfalls, müssen im Rahmen der Implementierung der Schemamigration allerdings nicht angepasst werden.

Controller

Controller werden im RDF-Architect einzig für die Definition der API-Endpunkte verwendet und enthalten keine Geschäftslogik. Code 3.1 zeigt beispielhaft die Implementierung des Dataset-Controllers, und einen Endpunkt, der ein angegebenes Dataset löscht.

```
@RestController
@RequestMapping("api/datasets")
@RequiredArgsConstructor
public class DatasetRESTController {
    private final DeleteDatasetUseCase deleteDatasetUseCase;

    @Operation(...) // OpenAPI-Dokumentation des Endpunkts
    @DeleteMapping("/{datasetName}")
    public String deleteDataset(
              @RequestHeader(...) String originURL,
              @PathVariable String datasetName) {
        deleteDatasetUseCase.deleteDataset(datasetName);
        return "success";
    }

    // Weitere Endpunkte...
}
```

Code 3.1: Beispiel eines Controllers im RDF-Architect

Die Klasse ist mit der Annotation `@RestController` versehen, die sie als REST-Controller kennzeichnet. Durch diese Kennzeichnung übernimmt Spring Boot automatisch die Initialisierung der Klasse und erlaubt zudem die Zuweisung einer Route über die Annotation `@RequestMapping`. In diesem Fall wird der Controller für alle Anfragen unterhalb von `api/datasets` zuständig sein. Endpunkte, die wie in diesem Beispiel weitere Identifier benötigen, können die Basis-Route weiter modifizieren. Ein weiterer Vorteil durch die Nutzung der `@RestController`-Annotation ist, dass

Rückgabewerte der Methoden automatisch durch Jackson zu JSON serialisiert und als Response-Body versendet werden.

Zudem hat die Klasse noch die `@RequiredArgsConstructor`-Annotation. Hierbei handelt es sich um eine Lombok-Annotation, die automatisch einen Konstruktor generiert, der alle finalen Felder der Klasse als Parameter entgegennimmt. Dieser Konstruktor kann dann von Spring Boot genutzt werden, um automatisch eine Instanz des Controllers mit seinen Abhängigkeiten zu erstellen und diese zu injizieren.

Innerhalb der Klasse wird ein Endpunkt zum Auflisten aller Datasets als Methode definiert. Diese Methode ist mit der `@DeleteMapping`-Annotation versehen, die sie als Handler für DELETE-Anfragen kennzeichnet. Weitere HTTP-Methoden wie GET, POST, oder PUT können durch entsprechende Annotationen wie `@GetMapping`, `@PostMapping` und `@PutMapping` implementiert werden. Mögliche URL-Parameter, Request-Header oder Request-Bodies können durch entsprechende Annotationen als Parameter der Methode definiert werden. In diesem Fall wird der Name des zu löschenden Datasets als URL-Parameter übergeben und der Origin-Header der Anfrage als Request-Header.

Zusätzlich ist der Endpunkt mit einer OpenAPI-Annotation versehen. Die dort angegeben Informationen können von Tools wie Swagger genutzt werden, um automatisch eine API-Dokumentation zu generieren. Die genaue Implementierung dieser OpenAPI Informationen ist in dem Code Beispiel gekürzt, da sie für die Funktionalität des Controllers nicht relevant ist.

Inhalt der Methode ist lediglich das Aufrufen eines Interfaces, das die Geschäftslogik zum Löschen eines Datasets kapselt. Mögliche Exceptions, die während der Ausführung auftreten, werden automatisch von einem globalen Exception-Handler abgefangen und in entsprechende HTTP-Responses umgewandelt.

Models

Innerhalb des Models-Package werden die verschiedenen Datenmodelle der Anwendung definiert. Diese Modelle werden für die Arbeit mit Daten in den Services genutzt, wobei für die Kommunikation mit dem Frontend teilweise eigene vereinfachte DTO-Klassen (Data Transfer Objects) genutzt werden. Code 3.2 zeigt das Modell für die Repräsentation einer CIM-Klasse im RDF-Architect. Das Beispiel verdeutlicht vor allem, wie Lombok-Annotationen genutzt werden können, um Modelle kompakt zu definieren.

`@Data` fasst direkt mehrere Lombok-Annotationen zusammen und generiert automatisch Getter und Setter für alle Felder, sowie `equals()`, `hashCode()` und

`toString()` Methoden. Außerdem wird ein Konstruktor mit allen finalen Feldern der Klasse generiert. Zusätzlich wird durch den `@NoArgsConstructor` ein Konstruktor ohne Parameter erstellt, der für die Serialisierung mittels Jackson benötigt wird.

```
@Data
@NoArgsConstructor
@SuperBuilder(toBuilder = true)
public class CIMClass {
    private UUID uuid;
    private URI uri;
    private RDFSLabel label;
    private RDFSSubClassOf superClass;
    private RDFSComment comment;

    @JsonProperty("package")
    private CIMSBelongsToCategory belongsToCategory;

    @Builder.Default
    @JsonInclude(JsonInclude.Include.NON_NULL)
    private List<CIMSStereotype> stereotypes = new ArrayList<>();

    public void nullStereotypes() {
        if (stereotypes.isEmpty()) {
            stereotypes = null;
        }
    }
}
```

Code 3.2: Beispiel eines Datenmodells im RDF-Architect

`@SuperBuilder` ermöglicht schließlich noch die Nutzung des Builder-Patterns zum Erstellen von Instanzen, bei dem gezielt nur bestimmte Felder gesetzt werden können. Im Gegensatz zu der normalen `@Builder`-Annotation wird `@SuperBuilder` für Klassen mit Vererbungshierarchien genutzt. Im Zusammenhang mit der Builder-Annotation können bestimmte Felder mit `@Builder.Default` annotiert werden, um ihnen einen Default-Wert zuzuweisen, falls ihnen beim Erstellen durch den Builder kein Wert zugewiesen wird.

Die letzten beiden Annotationen `@JsonProperty` und `@JsonInclude` sind für die JSON-Serialisierung mit Jackson relevant, werden in dieser Form in der Arbeit jedoch nicht weiter genutzt.

Services

Services beinhalten die Geschäftslogik der Anwendung und implementieren die verschiedenen Use Case Interfaces, die von den Controllern genutzt werden. Code 3.3 zeigt den `UpdateDatasetService` und seine Methode `deleteDataset`.

```
@Service
@RequiredArgsConstructor
public class UpdateDatasetService implements AddPrefixUseCase,
    ReplacePrefixUseCase, DeletePrefixUseCase, DeleteDatasetUseCase {
    private final DatabasePort databasePort;

    @Override
    public void deleteDataset(String datasetName) {
        databasePort.deleteDataset(datasetName);
    }

    // Weitere Methoden...
}
```

Code 3.3: Beispiel eines Services im RDF-Architect

Services sind mit der `@Service`-Annotation versehen, die sie als Service-Klasse kennzeichnet. Ähnlich wie bei den Controllern erlaubt diese Annotation das automatische Initialisieren von Services und die Injektion der Abhängigkeiten durch Spring Boot. Hierfür wird ebenfalls mittels der `@RequiredArgsConstructor`-Annotation ein Konstruktor generiert, der alle finalen Felder der Klasse als Parameter entgegennimmt. Der Service hat als einziges Feld das Interface `DatabasePort`, wobei es sich um die Anbindung an die Datenbank handelt.

In diesem Beispiel enthält die Methode `deleteDataset` lediglich den Aufruf der entsprechenden Methode des `DatabasePort`-Interfaces, um das angegebene Dataset zu löschen. Andere Services können jedoch deutlich umfangreichere Logik enthalten oder auch weitere Utility-Klassen nutzen, um die Logik zu strukturieren.

3.3 Frontend

Das Frontend des RDF-Architects ist in Svelte 5 und Javascript geschrieben und nutzt das SvelteKit-Framework zum Routing und Bauen der Anwendung. Datenanfragen an das Backend erfolgen über klassische REST-API-Aufrufe, bei

denen JSON als Datenformat genutzt wird. Die UML-Diagramme der Modelle werden mithilfe der Bibliothek mermaid.js gerendert, welche aus einem textuellen Format SVG-Grafiken generiert und darstellt.

3.3.1 Projektstruktur

Der Source-Ordner des Frontends ist in zwei Hauptordner unterteilt: `/routes` und `/lib`. Der Routes-Ordner enthält die Svelte Komponenten für die verschiedenen Seiten der Anwendung, während der Lib-Ordner vor allem wiederverwendbare Komponenten und Hilfsfunktionen enthält.

SvelteKit nutzt File-based Routing, was bedeutet, dass die Struktur des Routes-Ordners die URL-Struktur der Anwendung widerspiegelt. Als Einstiegspunkt dient die `+page.svelte` Datei im Root-Verzeichnis des Routes-Ordners. Diese Datei wird beim Aufruf der Basis-URL der Anwendung geladen und rendert die Hauptkomponente der Anwendung. Weitere Seiten werden durch das Anlegen von Unterordnern erstellt, wobei jeder dieser Unterordner ebenfalls eine `+page.svelte` Datei enthalten muss, die die jeweilige Seite rendert.

Neben der `+page.svelte` Datei im Root-Verzeichnis liegt im RDF-Architect zusätzlich eine `+layout.svelte` Datei, welche das zugrundeliegende Layout der Anwendung definiert. Diese Layout-Datei wird auf allen Seiten der Anwendung gerendert und enthält die Menu-Leiste, die für die Navigation zwischen den Seiten genutzt werden kann.

Der Code einer Seite muss nicht nur in der `+page.svelte` Datei liegen, sondern kann in weitere Komponenten ausgelagert werden, die dann von den Seiten importiert und genutzt werden. Komponenten, die von mehreren Seiten genutzt werden, liegen dabei im `/lib/components` Ordner, während Seiten-spezifische Komponenten im jeweiligen Seitenordner liegen.

3.3.2 Aufbau einer Svelte-Datei

Eine Svelte-Datei kann aus drei verschiedene Bereiche bestehen: dem Script-Bereich, der Javascript Code enthält, dem Template-Bereich, der das HTML-Template der Komponente definiert, und dem Style-Bereich, der CSS-Styles für die Komponente enthält. Der Style Bereich wird im RDF-Architect allerdings nicht genutzt, da die Styles per Tailwind-CSS auf den HTML-Elementen definiert werden.

Reaktive Variablen

Der Script-Bereich enthält, wie auch in anderen Javascript-Frameworks, Javascript-Methoden, Variablendeklarationen und Lifecycle-Hooks der Komponente. Statt Variablen nur mit `let` und `const` zu deklarieren, können Variablen in Svelte zusätzlich als State-Variablen deklariert werden. State-Variablen sind reaktive Variablen, und HTML-Elemente, die auf diese Variablen zugreifen, werden automatisch neu gerendert, wenn sich der Wert der Variable ändert.

State-Variablen müssen allerdings immer noch manuell neue Werte zugewiesen bekommen. Im Gegensatz dazu stehen Derived-Variablen. Diese Variablen können bei ihrer Deklaration eine Funktion zugewiesen bekommen, anhand derer sich ihr Wert automatisch updatet.

Eine weitere Möglichkeit, Werte reaktiv neu zu berechnen, ist der sogenannte Effect-Block. Diese Blöcke enthalten ebenfalls Funktionen, die automatisch neu ausgeführt werden, sobald sich einer der genutzten Werte ändert. Dadurch lassen sich mehrere Werte gleichzeitig neu berechnen, oder auch Seiteneffekte auslösen, wie das Schreiben eines Logs in die Konsole.

Templates

Der Template-Bereich einer Svelte-Datei ist ähnlich aufgebaut wie in anderen Frameworks. HTML-Blöcke können durch Svelte-spezifische Syntax erweitert werden, um beispielsweise Schleifen oder Bedingungen zu nutzen. Svelte-Direktiven werden dabei meistens mit einem `#` Symbol eingeleitet, wie beispielsweise `#if` für Bedingungen oder `#each` für Schleifen.

Neben den Standard-HTML-Elementen können im Template auch selbst definierte Svelte-Komponenten eingebunden werden.

3.3.3 Teilen von Variablen zwischen Komponenten

Variablen können in Svelte auf verschiedene Arten zwischen Komponenten geteilt werden. Falls ein Wert nur von einer Elternkomponente an ihre Kindkomponente weitergegeben werden soll, kann dies über sogenannten Props erfolgen.

Eine Komponente kann beliebig viele ihrer Variablen als Props deklarieren, sodass diese beim Einbinden der Komponente gesetzt werden können. Dabei handelt es sich um eine Referenz des Wertes, wodurch Änderungen in der Kindkomponente zu einem gewissen Grad auch die Variable in der Elternkomponente ändern können. Wird beispielsweise eine Liste als Prop übergeben, kann die Kindkomponente

Elemente dieser Liste ändern und die Elternkomponente sieht diese Änderungen ebenfalls. Wird die Liste allerdings komplett neu zugewiesen, ändert sich die Variable in der Elternkomponente nicht. In diesem Fall muss das Prop zusätzlich als `bindable` deklariert werden, wodurch auch Änderungen der Identität des Wertes in der Kindkomponente in der Elternkomponente reflektiert werden.

Eine weitere Methode, Variablen zwischen Komponenten zu teilen, sind Svelte Stores. Diese Svelte Stores sind globale, reaktive Variablen, und können daher genutzt werden, um Werte zwischen beliebigen Komponenten der Anwendung zu teilen, ohne dass diese in einer Eltern-Kind-Beziehung stehen müssen.

4 Konzeptionierung

Um die Schemamigration erfolgreich umsetzen zu können, ist eine sorgfältige Planung und Konzeptionierung des gewünschten Verhaltens notwendig. Dieses Kapitel beschreibt die erarbeiteten Konzepte und Entwürfe, die als Grundlage für die Implementierung dienen.

Abschnitt 4.1 gibt auf Basis der in Unterabschnitt 2.2.3 vorgestellten Struktur der CIM-Ressourcen einen detaillierten Überblick über die verschiedenen Änderungen, die in einem Schema-Update auftreten können, und welche Auswirkungen diese auf die Instanzdaten haben. In Abschnitt 4.2 werden bestehenden Ansätze zur Umbenennungserkennung vorgestellt und bewertet, um eine geeignete Methode für die Erweiterung des RDF-Architect zu finden. Aufgrund dieser erarbeiten Umbenennungserkennung und den generellen Anforderungen werden in Abschnitt 4.3 die benötigten Migrationsschritte im Front- und Backend beschrieben. Abschließend wird in Abschnitt 4.4 die Konzeptionierung des User Interfaces beschrieben, welches den Nutzer durch den Migrationsprozess führen soll.

4.1 Analyse der möglichen Schemaänderungen

Dieser Abschnitt stellt alle möglichen Änderung vor, die aus einem Schema-Update resultieren können, und welche Auswirkungen sie auf die Instanzdaten haben. Diese Änderungen lassen sich anhand der in Abschnitt 2.2.3 vorgestellten Syntax der CIM-Ressourcen ableiten.

Generelle Änderungen

Neben Änderungen an bestimmten Eigenschaften von Ressourcen lassen sich auch Änderungen an den Ressourcen selbst identifizieren; eine Ressource kann gelöscht oder hinzugefügt werden. Das Löschen einer Ressource hat immer eine klare Konsequenz: alle Instanzen, die diese Ressource instanziieren, müssen ebenfalls gelöscht werden. Das Hinzufügen einer neuen Ressource kann hingegen verschiedene Auswirkungen haben. Im Fall von Klassen oder Enum Entries erfordert die Änderung keine Aktionen, Pflicht-Attribute und -Assoziationen müssen jedoch in den Instanzen der entsprechenden Klassen initialisiert werden.

Generell wird davon ausgegangen, dass die geänderte Version des Schema mithilfe des RDF-Architects erstellt wurde und daher inhaltlich vollständig ist. Zum Beispiel

wird davon ausgegangen, dass, wenn eine Klasse im Rahmen der Schemaupdates gelöscht wurde, alle Assoziationen zu dieser Klasse ebenfalls in der neuen Version entfernt wurden. Dadurch müssen als Aktion für die gelöschte Klasse nur alle Instanzen gelöscht werden; mögliche Seiteneffekte werden an anderer Stelle behandelt.

CIM-Packages

Da Packages lediglich genutzt werden, um Klassen in dem Schema-Graph zu gruppieren, und nicht in den Instanzen widergespiegelt werden, können alle Änderungen ignoriert werden.

CIM-Klassen

Das `rdf:type` Tripel einer Klasse dient der Identifizierung der Ressource als Klasse und kann daher ignoriert werden. Da dieses Tripel immer den Wert `rdfs:Class` haben muss, können wir davon ausgehen, dass keine Änderungen vorliegen können. Auch Änderungen an `rdfs:comment` und `cims:belongsToCategory` können ignoriert werden, da Kommentare und Package-Informationen nur im Schema liegen.

Das `rdfs:label` muss etwas differenzierter betrachtet werden. Es existiert im Schema um einen kürzeren und besser lesbaren Namen zuzuweisen, gegenüber der IRI, die zur eigentlichen Identifikation genutzt wird. Demnach wird das Label auch nicht in die Instanzdaten übertragen. Allerdings geht mit einer Änderung des Labels in des meisten Fällen auch eine Änderung der IRI einher, da diese üblicherweise das Label enthält. In diesem Fall muss daher die Umbenennung richtig erkannt werden, damit Instanzdaten nicht gelöscht werden.

Eine Änderung an `subClassOf` hat ebenfalls indirekte Konsequenzen, da sich gegebenenfalls die geerbten Properties geändert haben. Es müssen also alle vorher geerbten Properties, die nun nicht mehr geerbt werden, entfernt werden, und alle neuen Properties auf den Instanzen hinzugefügt werden.

Zuletzt ist noch ein spezifischer Stereotyp relevant: `http://iec.ch/TC57/NonStandard/UML#concrete`. Dieser Stereotyp steuert, ob eine Klasse instanziiert werden kann. Falls er entfernt wird, müssen demnach wie beim Löschen der gesamten Klasse alle Instanzen der Klasse gelöscht werden. Alle anderen Stereotypenänderungen können ignoriert werden, da diese keine Auswirkungen auf die Instanziierbarkeit haben.

Tabelle 4.1 fasst die für die Instanzdaten relevanten Klassenänderungen und die entsprechenden Migrationsaktionen zusammen.

Änderungstyp	benötigte Migration
Klasse gelöscht	alle Instanzen der Klasse löschen
Label/URI geändert	type Prädikat der Instanzen ändern
SubClass Relation geändert	alte geerbten Properties löschen und neue hinzufügen
concrete Stereotype gelöscht	alle Instanzen der Klasse löschen

Tabelle 4.1: Klassenänderungen

CIM-Attribute

Ähnlich wie bei den Klassen können `rdf:type` und `cims:stereotype` ignoriert werden, da diese Prädikate jeweils feste Werte haben. Auch Kommentar und Label verhalten sich hier analog.

Eine Änderung der Domain verhält sich ähnlich wie eine Label Änderung, da auch hier die IRI geändert wird. Hier sollte demnach auch eine Zuordnung der alten und neuen Ressource stattfinden. Allerdings kann die IRI in diesem Fall nicht einfach angepasst werden, da das Attribut durch die Domain-Änderung nun auf einer neuen Klasse definiert ist. Das Attribut muss dementsprechend auf allen Instanzen der alten Domain entfernt und auf Instanzen der neuen Domain hinzugefügt werden. Dabei müssen ebenfalls erbende Klassen in Betracht gezogen werden.

Bei Änderungen des Datentyps sollten bestehende Werte, wenn möglich, zu diesem neuen Datentyp konvertiert werden. Ist die Konvertierung nicht möglich, sollte stattdessen ein Default-Wert gesetzt werden. Hat das Attribut ein Enum als Datentyp, sollte einer der Enum-Werte ausgewählt und als neuer Wert gesetzt werden.

Wie der `concrete` Stereotyp bei den Klassen gibt die Multiplicity bei den Attributen Auskunft über die Instanziierbarkeit. Wird ein Attribut optional, hat dies keinen Einfluss auf bestehende Instanzen des Attributs. Wird ein vorher optionales Attribut jedoch zu einem Pflichtfeld, muss es in allen Instanzen der Domain und erbenden Klassen mit einem Default-Wert hinzugefügt werden. Dabei sollten bestehende Werte des Attributs allerdings nicht mit dem Default-Wert überschrieben werden.

`cims:isDefault` und `cims:isFixed` sind nur zu gewissen Teilen für die Instanzdaten relevant. Ein Hinzufügen, Ändern oder Entfernen von `cims:isDefault` hat

keinen Einfluss auf bestehende Instanzen, da die Instanzen bereits konkrete Werte haben. Das Hinzufügen oder Ändern von `cims:isFixed` hat hingegen die Konsequenz, dass alle Instanzen des Attributs auf den neuen Fixed-Wert gesetzt werden müssen.

Tabelle 4.2 fasst die für die Instanzdaten relevanten Attributänderungen und die entsprechenden Migrationsaktionen zusammen.

Änderungstyp	benötigte Migration
Attribut gelöscht	Attribut auf allen Instanzen löschen
Pflicht-Attribut hinzugefügt	Attribut zu Instanzen der Domain mit Default-Wert hinzufügen
Label/URI geändert	Prädikat-IRI auf Instanzen ändern
Domain geändert	Attribut auf alten Instanzen löschen und auf neuen hinzufügen
Datatype geändert	alten Wert wenn möglich konvertieren oder Default-Wert einsetzen
Range geändert	Wert durch einen Default-Wert des neuen Enums ersetzen
zu Pflicht-Attribut gemacht	Attribut, wenn noch nicht vorhanden, zu Instanzen der Domain mit Default-Wert hinzufügen
isFixed hinzugefügt/geändert	Instanzdaten auf Fixed-Wert setzen

Tabelle 4.2: Attribut-Änderungen

CIM-Assoziationen

Änderungen von bekannten Prädikaten verhalten sich auch hier analog zu den anderen Ressourcen.

`rdfs:range` verhält sich ähnlich zu der Domain, da auch hier die IRI angepasst wird, allerdings können in diesem Fall nicht die alten Werte durch eine Umbenennungserkennung beibehalten werden. Durch die Änderung der Ziel-Klasse müssen hier komplett neue Werte für die Assoziation angegeben werden.

Änderungen von `cims:inverseRoleName` müssen nicht beachtet werden, da diese nur im Schema genutzt werden, um leichter die inverse Assoziation zu finden.

Mit `cims:associationUsed` gibt es erneut ein Prädikat, das Auskunft über die Instanziierbarkeit der Assoziation gibt. Bei einer Änderung zu „Nein“ müssen daher bisherige Instanzen gelöscht werden, eine Änderung zu „Ja“ erfordert jedoch keine Aktion.

Auch die Multiplicity gibt in gewisser Weise Auskunft über die Instanziierbarkeit, allerdings ist die Situation hier etwas komplexer als bei den Attributen. Eine Erhöhung der minimalen Multiplicity erfordert das Hinzufügen von neuen Assoziationen zu Instanzen der Domain, um die neue Mindestanzahl zu erfüllen. Ein Verringern der maximalen Multiplicity erfordert hingegen das Löschen von überzähligen Assoziationen auf Instanzen der Domain.

Allerdings können beim Hinzufügen von neuen Assoziationen, durch Änderungen der Multiplicity oder von `cim:associationUsed`, nicht einfach wie bei Attributen Default-Werte gesetzt werden, da die Ziel-Instanzen der Assoziation nicht bekannt sind. Um dennoch sinnvolle Default-Werte generieren zu können, kann ein SPARQL-Pattern genutzt werden. Dieses Pattern würde aus einem vom Nutzer angegebenen WHERE-Block bestehen, der beim Ausführen des Migrationsskriptes Instanzen findet und einfügt.

Tabelle 4.3 fasst die für die Instanzdaten relevanten Assoziationsänderungen und die entsprechenden Migrationsaktionen zusammen.

CIM-Enum-Entries

Enum Entries haben außer ihrem Label und der IRI keine weiteren Eigenschaften, die gesondert beachtet werden müssen. Wie auch bei anderen Ressourcen müssen lediglich Umbenennungen und Löschungen erkannt und behandelt werden. Im Falle einer Löschung bedeutet dies, dass ein neuer Default Enum-Entry der Enum-Klasse ausgewählt werden muss, der den gelöschten Wert in allen Attributen ersetzt, die die Enum-Klasse als Datentyp nutzen.

Tabelle 4.4 fasst die für die Instanzdaten relevanten Enum-Entry-Änderungen und die entsprechenden Migrationsaktionen zusammen.

4.2 Umbenennungserkennung

Zentrale Motivation für die Implementierung einer automatisierten Schemamigration im RDF-Architect ist die Erhaltung von Daten bei Änderungen am Schema. Dies ist besonders relevant bei Umbenennungen von Klassen und Properties. Da

Änderungstyp	benötigte Migration
Assoziation gelöscht	Assoziation auf allen Instanzen löschen
Pflicht-Assoziation hinzugefügt	benötigte Anzahl an Assoziationen zu Instanzen der Domain hinzufügen
Label/IRI geändert	Prädikat-IRI auf Instanzen ändern
Domain geändert	Assoziationen auf alten Instanzen löschen und auf neuen hinzufügen
Range geändert	alte Assoziation löschen und Assoziation mit neuer Range hinzufügen
minimale Multiplicity erhöht	fehlende Anzahl an Assoziationen zu Instanzen der Domain hinzufügen
maximale Multiplicity verringert	überzählige Assoziationen auf Instanzen der Domain löschen
associationUsed zu Nein geändert	alle Instanzen der Assoziation löschen

Tabelle 4.3: Assoziation-Änderungen

Änderungstyp	benötigte Migration
Enum-Entry gelöscht	in Instanzen durch neuen Default-Wert der Enum-Klasse ersetzen
Label/URI geändert	Wert für Attribute ändern

Tabelle 4.4: Enum-Entry-Änderungen

Ressourcen in RDF-Graphen über ihre IRI referenziert werden, würde eine Umbenennung ohne robuste Erkennung davon dazu führen, dass alle Instanzen der umbenannten Ressource gelöscht werden müssten. Um dies zu verhindern, muss die Schemamigration in der Lage sein, solche Umbenennungen zu identifizieren und im Migrationsskript entsprechend zu berücksichtigen.

In den folgenden Abschnitte werden zunächst bestehende Ansätze zur Erkennung von Umbenennungen in Schemata untersucht und daraufhin ein eigener Ansatz zur Umbenennungserkennung in CIM-Schemata vorgestellt.

4.2.1 Analyse bestehender Lösungen

Ein großes bestehendes Forschungsfeld in der Literatur ist das sogenannte „Ontology Alignment“. Hierbei sollen verschiedenen Modelle der gleichen Domäne verglichen und Klassen gefunden werden, die das gleiche Konzept beschreiben. Häufig stammen die Ontologien, die dadurch vereinigt werden sollen, aus dem Bereich der Medizin oder der IT.

Ardjani et al. [41] geben einen Überblick über bestehende Ansätze zum Ontology Alignment. Darin werden vier verschiedenen Kategorien identifiziert, nach denen sich die Umsetzungen gruppieren lassen.

- terminologisch: Berechnung der Ähnlichkeit von Zeichenketten wie Label, Kommentare, Beschreibungen etc. Diese können zusätzlich aufgeteilt werden in syntaktischen und semantischen Vergleich der Zeichenketten.
- strukturell: Vergleich von internen Werten der Entitäten sowie Relationen zu externen Entitäten
- extensiv: Vergleich basierend auf Betrachtung der Instanzen
- semantisch: Nutzung einer gemeinsamen Referenz-Ontologie

Beinahe alle der analysierten Ansätze nutzen eine Form des terminologischen Vergleichs, während nur einige Strukturdaten in Betracht ziehen.

Viele der Konzepte des Ontology Alignments lassen sich auf die Umbenennungserkennung nach einem Schemaupdate übertragen. Für die Umbenennungserkennung bietet sich ein hybrider Ansatz aus terminologischem und strukturellen Vergleich an. Ein extensiver Ansatz ist nicht möglich, da zur Zeit der Erstellung des Migrationsskriptes keine Instanzen vorhanden sind, während für einen semantischen Ansatz eine gemeinsame Referenz-Ontologie fehlt.

4.2.2 Generelle Überlegungen - Scope der Umbenennungserkennung

Gegenüber dem Ontology Alignment bieten sich bei der Umbenennungserkennung nach einem Schemaupdate durch den enger gefassten Nutzungsrahmen einige Vorteile. Während beim Ontology Alignment alle Entitäten der beiden Ontologien verglichen und zugeordnet werden müssen, ist die Umbenennungserkennung nur auf die tatsächlich geänderten Ressourcen fokussiert. Es kann davon ausgegangen werden, dass jede Umbenennung immer aus einer gelöschten Ressource aus dem alten Schema und einer neu erstellten Ressource im neuen Schema besteht. Dadurch

reduziert sich die Anzahl der zu vergleichenden Ressourcen deutlich, was die Berechnungszeit verringert und die Genauigkeit der Ergebnisse erhöht.

Um die Anzahl der Vergleiche noch weiter zu reduzieren, kann zudem die Menge an zu vergleichenden Properties eingegrenzt werden. Statt alle gelöschten und hinzugefügten Properties zu vergleichen, kann nur ein Vergleich zwischen Properties der gleichen Domain durchgeführt werden. Es werden also beispielsweise nur Attribute verglichen, die sowohl im alten als auch im neuen Schema der Klasse Person zugeordnet sind. Dies verbessert ebenso die Performance und verhindert, dass häufige Attribute wie `id` oder `name`, die potentiell auf verschiedenen Klassen bearbeitet werden, fälschlicherweise als Umbenennung erkannt werden.

Diese Begrenzung der zu vergleichenden Properties hat jedoch auch den Nachteil, da ein beabsichtigter Wechsel der Domain eines Properties nicht erkannt werden kann. Dies wäre beispielsweise der Fall, wenn ein Attribut von einer Klasse auf deren Elternklasse verschoben wird. In diesen Fällen gehen die bestehenden Instanzdaten verloren und werden neu initialisiert. Da dies jedoch nur selten vorkommt, wird dieser Nachteil in Kauf genommen, um die generelle Performance und Genauigkeit der Umbenennungserkennung zu verbessern.

Ein weiterer Faktor, den es zu beachten gilt, ist die Zuordnung von Properties auf umbenannten Klassen. Wenn nur Properties vergleichen werden, die der gleichen Domain zugeordnet sind, entsteht eine Abhängigkeit zwischen Umbenennungserkennung der Klassen und der Properties. Alle Klassenumbenennungen müssen erst verifiziert und zusammengeführt werden, bevor die Properties dieser neuen zusammengeführten Klasse verglichen werden können. Daraus bedingt sich eine iterative Vorgehensweise, die in dem Design der Benutzeroberfläche in Abschnitt 4.4 berücksichtigt werden muss.

4.2.3 Terminologischer Vergleich

Der direkteste Indikator für eine Umbenennung einer Ressource ist die Ähnlichkeit des Labels. Wie auch bei vielen andere Verfahren im Bereich des Ontology Alignment kann daher ein terminologischer Vergleich dieser Labels durchgeführt werden, um Umbenennungen zwischen Schemaversionen zu erkennen.

Testaufbau

Um die Ähnlichkeit zweier Labels zu bestimmen, können die in Abschnitt 2.4 vorgestellten, und von Apache Commons Text implementierten, Stringvergleichsalgorithmen genutzt werden. Die bereits implementierte Tokengenerierung für die

Kosinus-Ähnlichkeit anhand einzelner Wörter kann hierbei nicht genutzt werden, da die Label immer nur aus einem einzigen Wort bestehen. Stattdessen werden die Label anhand von Bigrams in Token aufgeteilt und an die Kosinus-Distanz übergeben, deren Ergebnis daraufhin in einen Ähnlichkeitswert umgewandelt wird.

Um herauszufinden, welcher der Algorithmen sich am Besten für die Umbenennungserkennung eignet, sollen die Algorithmen anhand eines Beispieldatensatzes verglichen werden. Dafür wurde ein Testdatensatz mit synthetischen Labels erstellt, die zwar nicht direkt in CGMES-Datensätzen vorkommen, ihnen aber in ihrem Aufbau ähneln. Die Label sind jeweils CamelCase-Bezeichner und bestehen wie in den CGMES-Daten durchschnittlich aus zwei Wörtern, teilweise sind jedoch auch kürzere oder längere Bezeichner enthalten.

Jedem dieser Label wird eine Menge an möglichen Umbenennungskandidaten zugeordnet, sowie die für diesen Testfall korrekte Umbenennung. Tabelle 4.5 zeigt einen dieser Testfälle. Insgesamt wurden 35 verschiedene solcher Testfälle erstellt und eine vollständige Auflistung ist im Anhang unter Tabelle A.1 zu finden.

Ursprungslabel	Umbenennungs-kandidaten	tatsächliche Umbenennung	Umbenennungstyp
Equipment Statu	DeviceStatus, VoltageLevel, PhaseVoltage, Equipment, LineVoltage, PowerFactor, EquipmentStatus	EquipmentStatus	Tippfehler

Tabelle 4.5: Label Testfall

Die Umbenennungskandidaten für ein Label wurden gewählt, um einen möglichst realistischen Kontext für den Fall einer Umbenennungserkennung bei einem Schemaupdate abzubilden. Sie enthalten jeweils Labels, die syntaktisch und semantisch unabhängig von dem Ursprungslabel sind, allerdings durchaus Teilsequenzen davon enthalten können. Es sind jedoch keine Label enthalten, die eine Umstrukturierung des Ursprungslabels darstellen, da davon ausgegangen werden kann, dass in einem Datensatz nicht mehrere Labels enthalten sind, welche das genau gleiche Konzept beschreiben.

Die verschiedenen Testfälle wurden anhand der Art der Änderung, die bei einem Schemaupdate vorkommen könnten, in verschiedene Kategorien eingeordnet. Mögliche Änderungsarten sind:

- Tippfehler: Beheben eines einfachen Tippfehlers, entweder durch Ersetzen eines Zeichens oder Tauschen von zwei benachbarten Zeichen
- Tokenreihenfolge geändert: Änderung der Reihenfolge der Wörter in dem Camel-Case Bezeichner
- Abkürzung/Erweiterung: Ausschreiben oder Abkürzen eines Begriffes
- Suffix/Präfix geändert: Entfernen oder Hinzufügen eines Präfixes oder Suffixes
- Fachbegriff geändert: Ersetzen eines Begriffes durch ein Synonym oder einen ähnlichen Begriff

Auswertung

Abbildung 4.1 stellt dar, in wie vielen der 35 Test-Cases die Algorithmen die tatsächliche Umbenennung als am Wahrscheinlichsten bewertet haben. Dabei heben sich die Kosinus-, Jaccard- und Jaro-Winkler-Ähnlichkeit klar von den anderen Algorithmen ab.

Testfälle der Kategorie Tippfehler wurden von allen anderen Algorithmen zu 100% richtig zugeordnet, während Testfälle mit der Änderung eines Fachbegriffs von keinem Algorithmus korrekt erkannt wurden. Um solche semantischen Änderungen zu erkennen, müsste der reine Stringvergleich durch eine semantische Analyse ergänzt werden, beispielsweise durch die Nutzung von WordNet. Insgesamt haben alle Algorithmen größere Schwierigkeiten bei der Erkennung von Abkürzungen und Erweiterungen sowie Tokenreihenfolgenänderungen, da diese Änderungen zu größeren Unterschieden in den Strings führen.

Allerdings ist die korrekte Erkennung einer Umbenennung nicht die einzige Metrik, die für die Auswahl eines Algorithmus relevant ist. Schließlich muss nicht jede gelöschte Ressource im alten Schema Teil einer Umbenennung sein. Daher sollten die Algorithmen auch in der Lage sein, Ressourcen zu identifizieren, die nicht umbenannt, sondern lediglich gelöscht wurden. Zu diesem Zweck wurden die Algorithmen anhand weiterer Testfälle evaluiert, in denen kein Truth-Wert angegeben wurde.

Abbildung 4.2a, Abbildung 4.2b und Abbildung 4.2c zeigen Boxplots der Ähnlichkeitswerte der drei besten Algorithmen, aufgeteilt nach richtig erkannten Umbenennungen (TP), nicht erkannten Umbenennungen (FN: falscher Kandidat) und Ressourcen, die nicht umbenannt wurden (FN: keine Umbenennung). Es ist zu erkennen, dass die Jaccard-Ähnlichkeit zwar die beste Trefferquote geliefert hat, jedoch für alle drei Kategorien eine große Spannweite an Werten liefert. Jaro-Winkler hingegen liefert für richtig erkannte Umbenennungen zuverlässig hohe Werte, tut dies jedoch

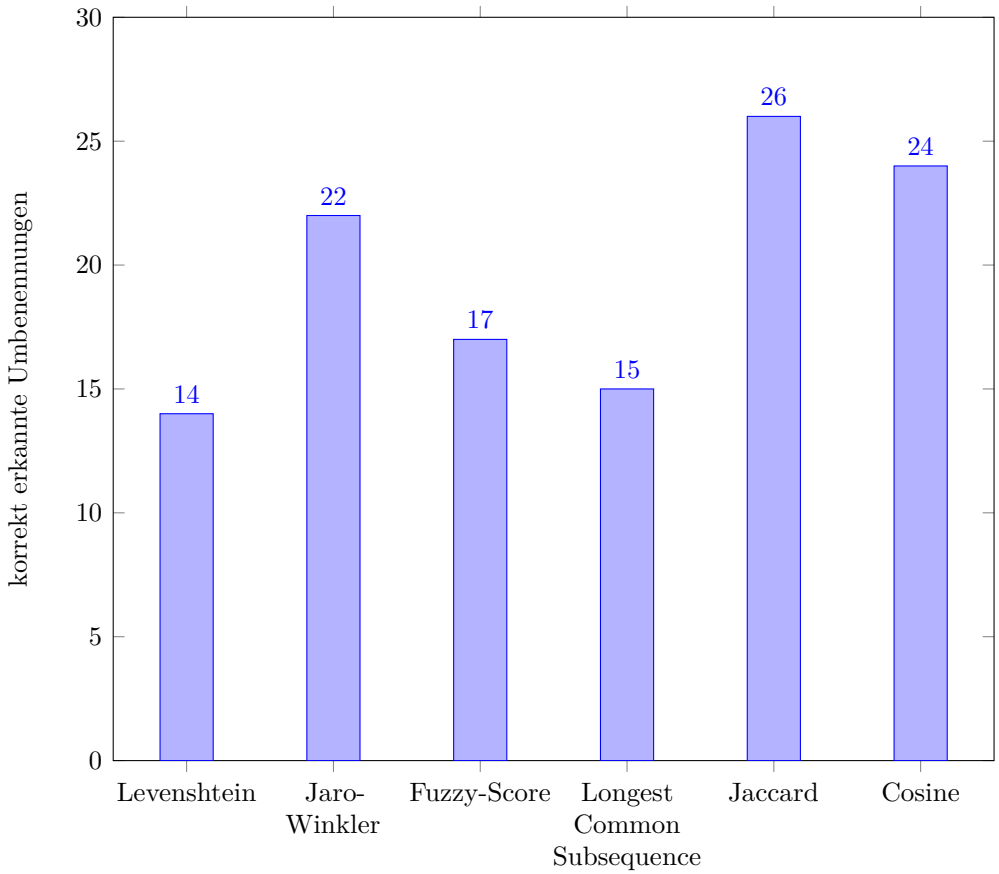
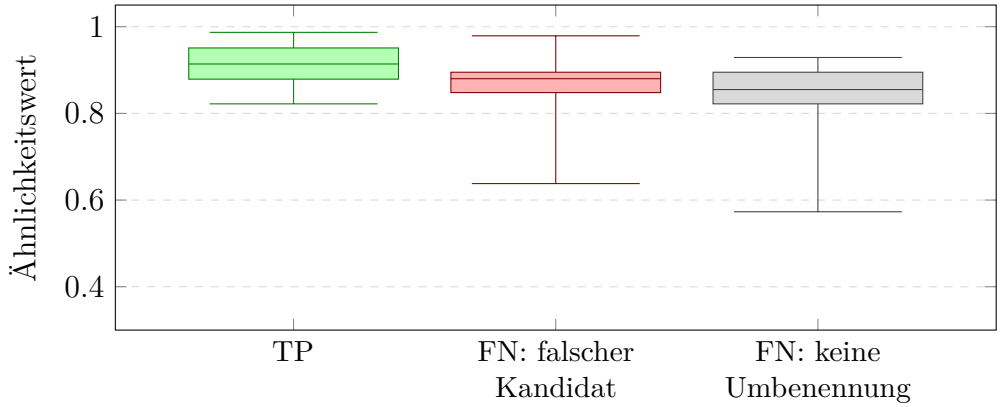


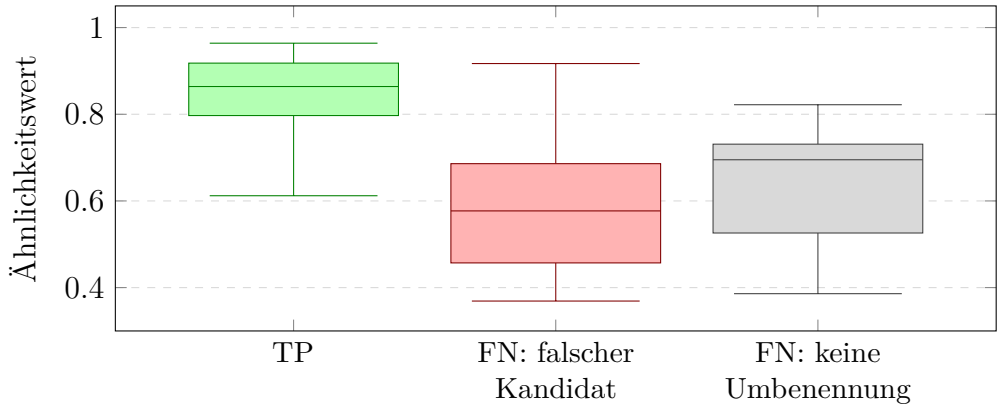
Abbildung 4.1: Vergleich der Algorithmen zur Umbenennungserkennung

auch für falsch erkannte Umbenennungen. Einzig die Kosinus-Ähnlichkeit zeigt eine deutliche Trennung zwischen richtig und falsch erkannten Umbenennungen.

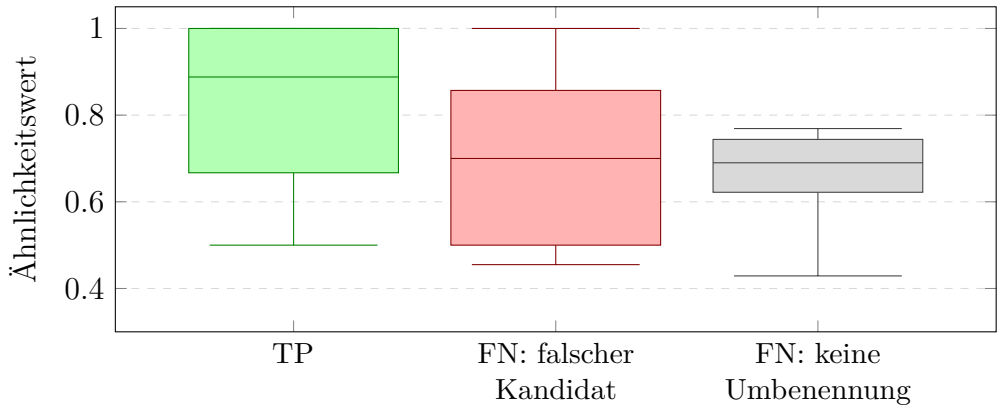
Diese klare Trennung ist besonders wichtig, um die Labelähnlichkeit mit anderen Indikatoren für eine Umbenennung kombinieren zu können, ohne das Ergebnis zu verfälschen. Daher ergibt sich aus dieser Analyse, dass die Kosinus-Ähnlichkeit mit Bigrams als Input der am besten geeignete Algorithmus ist, um Umbenennungen zu erkennen.



(a) Jaro-Winkler-Ähnlichkeit



(b) Kosinus-Ähnlichkeit



(c) Jaccard-Ähnlichkeit

Abbildung 4.2: Boxplots der Ähnlichkeitswerte der drei besten Algorithmen

4.2.4 Struktureller Vergleich

Durch die klare Struktur der CIM-Ressourcen, die in Unterabschnitt 2.2.3 vorgestellt wurde, lässt sich neben dem terminologischen Vergleich auch ein struktureller Vergleich durchführen. Dabei lassen sich für die verschiedenen CIM-Ressourcen verschiedene Properties vergleichen.

Bei einer Klasse können beispielsweise die Elternklasse, die zugehörigen Properties sowie die Stereotypen der Klasse verglichen werden. Im Gegensatz zu dem Label wird bei diesen Eigenschaften nicht mehr die lexikalische Ähnlichkeit der Werte in Betracht gezogen, sondern nur direkte Matches. Zwei Klassen gelten demnach beispielsweise als ähnlicher, wenn sie die selbe Elternklasse haben. Bei Mengen an Eigenschaften wie Properties oder Stereotypen wird die Anzahl der übereinstimmenden Properties addiert und durch die gesamte Anzahl der Properties geteilt.

Das Package der Klasse wird bewusst nicht verglichen, da in Schemata ab CGMES 3.0 alle Klassen in einem Package gespeichert werden. Ein Vergleich würde an dieser Stelle die Ähnlichkeit daher für jede beliebige Klasse erhöhen. Auch die Kommentare aller Ressourcen werden nicht verglichen, da eine genaue Überseinstimmung der Kommentare nach einer Umbenennung unwahrscheinlich ist, Kommentare zu verschiedenen Klassen aufgrund ähnlicher Formulierung allerdings fälschlicherweise eine lexikalische Ähnlichkeit aufweisen können.

Bei Attributen werden Datentyp, Multiplizität und jeweilige Default- und Fixed-Werte verglichen. Da Default- und Fixed-Werte optional sind, wird ein Fehlen dieser Werte auf beiden Attributen ebenfalls als eine Übereinstimmung gewertet. Die Domain wird nicht verglichen, da nur Attribute mit der gleichen Domain als Umbenennungen in Betracht gezogen werden.

Bei Assoziationen werden Target Klasse, Multiplicity und der AssociationUsed Wert verglichen. Auch hier wird die Domain aus dem gleichen Grund wie bei den Attributen nicht verglichen.

Enum Entries haben neben ihrer IRI keine weiteren strukturellen Informationen, die verglichen werden könnten. Daher ergibt sich die Ähnlichkeit zweier Enum-Entries alleine aus ihrer lexikalischen Ähnlichkeit.

Bei Klassen, Attributen und Assoziationen werden die verschiedenen Indikatoren für strukturelle Ähnlichkeit schließlich gewichtet und mit der lexikalischen Ähnlichkeit des Labels zu einem gesamtem Wert kombiniert. Die einzelnen Komponenten des strukturellen Vergleichs werden dabei untereinander gleich gewichtet.

4.2.5 Zusammenführen der Werte

Um die Umbenennungskandidaten untereinander zu vergleichen, müssen die lexikalische und strukturelle Ähnlichkeit zu einem aussagekräftigen Wert kombiniert werden. Die Analyse der Testfälle zeigt, dass die Kosinus-Ähnlichkeit in 69% der Fälle, in denen eine Umbenennung vorliegt, den richtigen Kandidaten als am Wahrscheinlichsten errechnet. Die Umbenennungserkennung anhand des Labels ist also größtenteils erfolgreich, sollte jedoch durch die strukturelle Ähnlichkeit unterstützt werden. Daher wurde für die Labelähnlichkeit ein Gewicht von 0.7 und für die strukturelle Ähnlichkeit ein Gewicht von 0.3 gewählt. Diese Gewichtung stellt sicher, dass die Labelähnlichkeit den Hauptindikator für eine Umbenennung darstellt, während die strukturelle Ähnlichkeit als unterstützender Faktor dient, um Fälle zu identifizieren, in denen die Labelähnlichkeit allein nicht ausreicht.

Allerdings müssen wie bereits erwähnt auch die nicht vorhandenen Umbenennungen sicher identifiziert werden. Zu diesem Zweck wird ein Grenzwert festgelegt, ab welchem ein Kandidat als Umbenennung in Betracht gezogen wird. Erreicht keiner der Kandidaten diesen Grenzwert, liegt keine Umbenennung vor und die Klasse wird als gelöscht betrachtet. Abbildung 4.2b zeigt, wie die Ähnlichkeitswerte der Kosinus-Ähnlichkeit und richtig erkannte Umbenennungen, falsch erkannte Umbenennungen und Löschungen ausfallen. Man kann erkennen, dass die richtig erkannten Umbenennungen größtenteils eine Ähnlichkeit von über 80% haben, während die andere beiden Kategorien klar separiert ab ca 75% anfangen. Der gewählte Grenzwert sollte demnach auch zwischen 80% und 75% liegen.

Tabelle 4.6 zeigt die jeweiligen richtig erkannten Umbenennungen (TP), als Umbenennung erkannten Löschungen (FP), richtig erkannten Löschungen (TN) und die falsch oder als Löschung erkannten Umbenennungen (FN) bei verschiedenen Grenzwerten. Bei einem Grenzwert von 85% werden alle Löschungen richtig erkannt, es erreichen jedoch auch viele der Umbenennungen nicht den benötigten Grenzwert. Bei einem Grenzwert von 70% hingegen erreichen zwar mehr Umbenennungen den Grenzwert, es werden allerdings auch beinahe die Hälfte aller Löschungen als Umbenennungen erkannt. Da davon ausgegangen werden kann, dass die gesamte Ähnlichkeit unter Anbetracht der strukturellen Informationen noch besser separiert ist als die Labelähnlichkeit alleine, bietet sich demnach 80% als sinnvoller Grenzwert an.

4.3 Migrationsablauf

Anhand der Überlegungen aus Abschnitt 4.2 und den generellen Anforderungen an die Schemamigration lassen sich die benötigten Schritte für den Migrationsablauf

Grenzwert	TP	FP	TN	FN
85%	13	0	10	22
80%	18	2	8	17
75%	18	2	8	17
70%	21	4	6	14

Tabelle 4.6: Kosinus-Ähnlichkeit Ergebnisse nach Grenzwert

ableiten. Abbildung 4.3 zeigt den geplanten Ablauf und welche Schritte dabei im Frontend und Backend ausgeführt werden müssen.

Um den Migrationsprozess zu starten, muss der Nutzer zunächst die alte und neue Version des Schemas angeben, zwischen denen migriert werden soll. Diese Schemata werden daraufhin an das Backend gesendet, welches die Schemata analysiert und auf Basis der geänderten Tripel die semantischen Änderungen zwischen den Versionen ableitet. Anhand der erkannten Änderungen können nun bereits die wahrscheinlichen Klassenumbenennungen bestimmt werden. Wie in Abschnitt 4.2 beschrieben, können in diesem Schritt Umbenennungen der Properties noch nicht erkannt werden, da vorher die Zusammenführung der umbenannten Klassen durchgeführt werden muss.

Daher werden vorher die potentiellen Klassenumbenennungen an das Frontend zurückgesendet, wo der Nutzer diese überprüfen und anpassen kann. Anschließend werden die bestätigten Klassenumbenennungen wieder an das Backend gesendet, welches die umbenannten Klassen in den Schemata zusammenführt. Mit den validierten Klassenumbenennungen können nun auch die Property-Umbenennungen errechnet werden. Diese werden ebenfalls an das Frontend zurückgesendet, wo der Nutzer diese überprüfen und anpassen kann.

Abschließend muss der Nutzer die benötigten Default-Werte angeben und diese an das Backend senden. Nachdem die Default-Werte empfangen wurden, hat das Backend nun alle Informationen, um das finale Migrationsskript zu generieren und dieses dem Nutzer zum Download anzubieten.

4.4 User Interface

Die Benutzeroberfläche der Schemamigration soll dem Benutzer eine einfache und übersichtliche Möglichkeit bieten, die Migration zu konfigurieren und durchzuführen. Wie in Unterabschnitt 4.2.2 beschrieben, muss die Migration iterativ

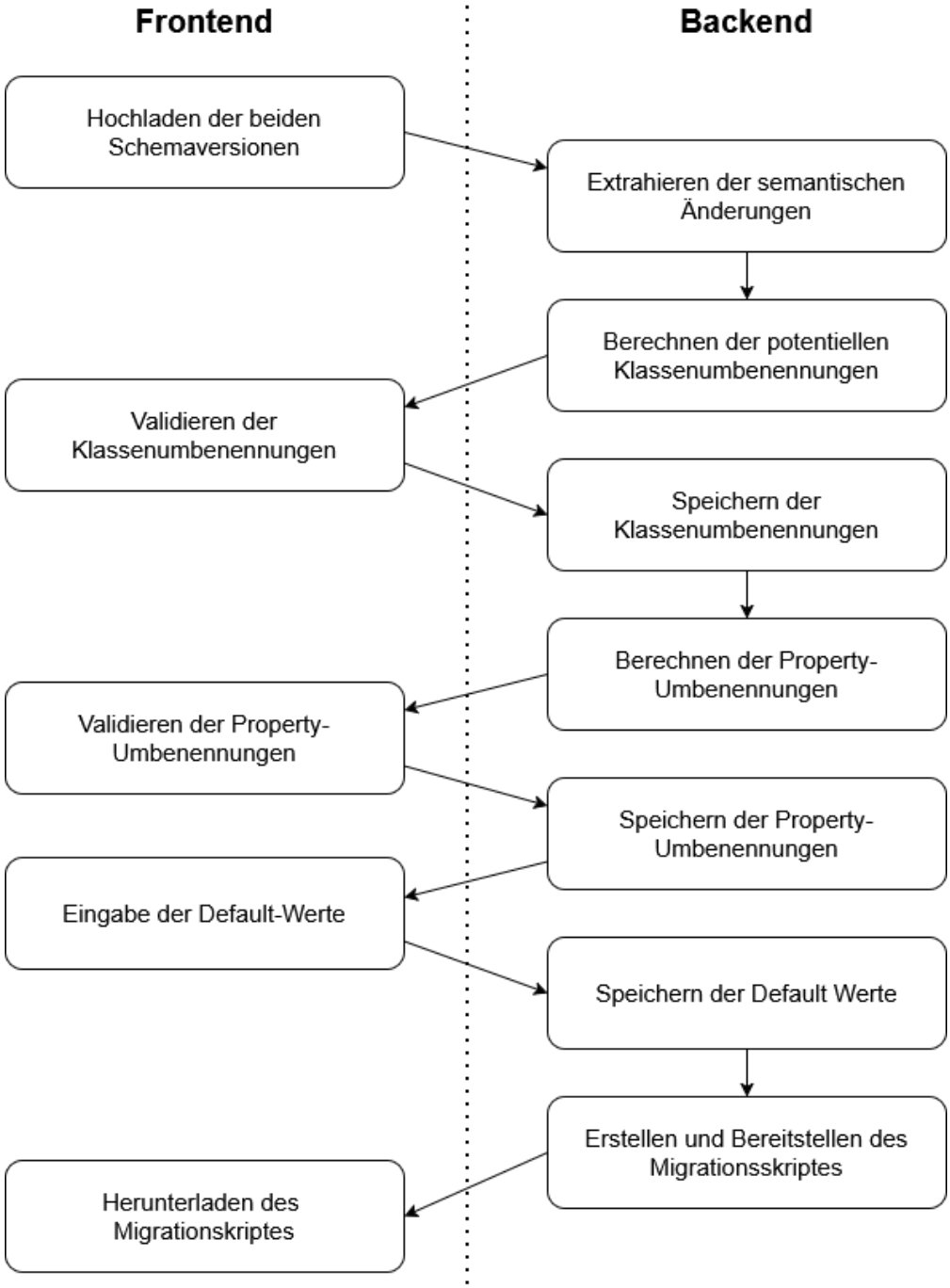


Abbildung 4.3: Migrationsablauf

durchgeführt werden, da Klassenumbenennungen bestätigt werden müssen, bevor die Property-Umbenennungen bestimmt werden können. Daher sollte eine Form der Darstellung gefunden werden, die diese iterative Vorgehensweise ermöglicht und sogar begünstigt.

Die Darstellung soll dabei möglichst intuitiv und benutzerfreundlich gestaltet sein, um den Nutzer durch den komplexen Prozess der Schemamigration zu führen. Gleichzeitig darf die Oberfläche nicht überladen wirken, und die Eingabe der benötigten Daten darf nicht zu langwierig sein.

4.4.1 Nutzung eines Web-Wizards

Für die Umsetzung der Benutzeroberfläche bietet sich die Nutzung eines Web-Wizards an. Dabei handelt es sich um ein UI-Design Pattern, welches häufig bei Programminstallationen verwendet wird, allerdings auch im Web eingesetzt wird, beispielsweise bei Checkout-Prozessen bei Online-Shops. Der Wizard ist darauf ausgelegt, komplexe Prozesse übersichtlich darzustellen. Dabei unterteilt der Wizard den Prozess in einzelne Schritte, durch welche der Benutzer nacheinander geführt wird. Jeder einzelne Schritt kann dabei übersichtlich gestaltet werden und dem Benutzer durch detaillierte Anweisungen helfen, die benötigten Daten einzugeben.

Um den Nutzer nicht zu überfordern, muss eine Balance zwischen möglichst wenigen Schritten und einer übersichtlichen Darstellung gefunden werden. Dabei hilft es, dem Benutzer eine Übersicht über die verbleibenden Schritte zu geben, damit er seinen Fortschritt abschätzen kann. Zudem sollte der Nutzer die Möglichkeit haben, zu vorherigen Schritten zurückzukehren, um Eingaben zu korrigieren.

Die Nutzung eines Web-Wizards bietet sich in dem Fall der Schemamigration besonders an, da die verschiedenen Schritte der Migration Abhängigkeiten zueinander haben und in einer bestimmten Reihenfolge ausgeführt werden müssen. Zudem handelt es sich durch den hohen benötigten Input des Nutzers um einen komplexen Prozess, der durch die Aufteilung in einzelne Schritte übersichtlicher gestaltet werden kann.

4.4.2 Gestaltung des Wizards

Die einzelnen Schritte des Wizards lassen sich anhand der bereits in Abschnitt 4.3 beschriebenen Migrationsschritten ableiten. Allerdings werden die beiden Schritte zur Validierung der Property-Umbenennungen und der Eingabe der Default-Werte nochmals in Unterschritte für die einzelnen Ressourcentypen unterteilt, um die Komplexität der einzelnen Schritte zu reduzieren.

4.4.3 Layout

Neben dem Inhalt der einzelnen Schritte, muss ein konsistentes Layout für den Wizard entworfen werden, insbesondere um die Navigation zwischen den Schritten zu ermöglichen.

Abbildung 4.4 zeigt den grundsätzlichen Aufbau des Wizards, der zwischen den verschiedenen Schritten bestehen bleibt. Am oberen Rand der Seite befindet sich eine Fortschrittsanzeige, die eine Übersicht über die verbleibenden Schritte gibt. Der aktuelle Schritt wird dabei farblich hervorgehoben. Falls der Schritt noch weitere Unterschritte hat, wie bei der Validierung der Property-Umbenennungen, wird eine zweite, kleinere Fortschrittsanzeige darunter angezeigt, die die Unterschritte darstellt. Auch hier wird der aktuelle Unterschritt visuell hervorgehoben.

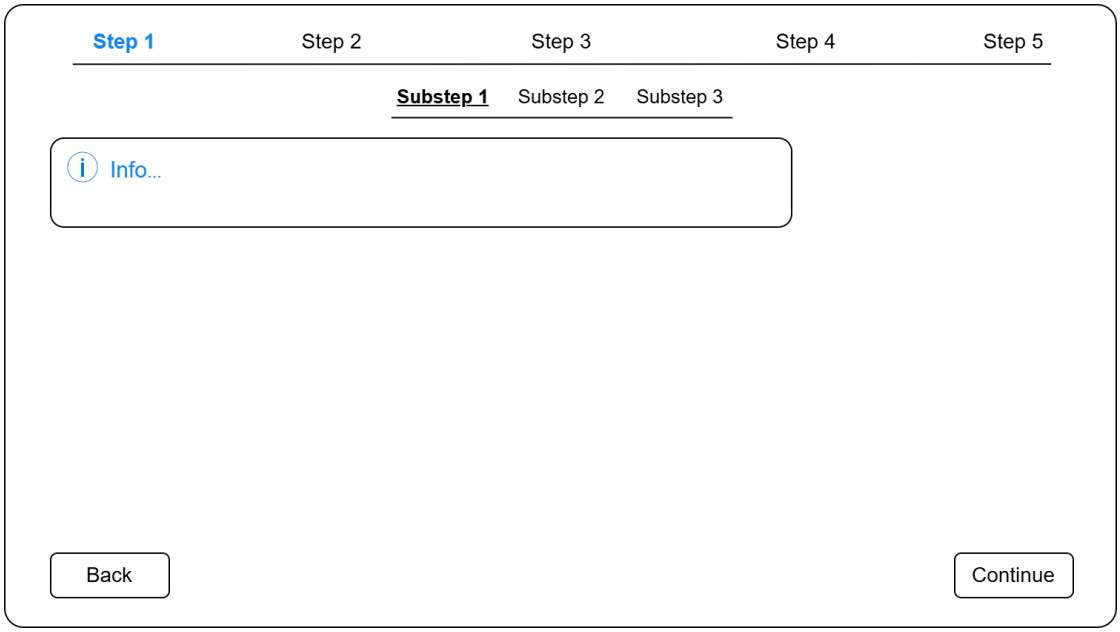


Abbildung 4.4: Aufbau des Migrations-Wizards

In der unteren linken und rechten Ecke befinden sich die Navigations-Buttons, mit denen der Nutzer zwischen den Schritten wechseln kann. Der Continue-Button ist dabei nur aktiv, wenn alle erforderlichen Eingaben in dem aktuellen Schritt gemacht wurden und wird ansonsten ausgegraut dargestellt.

Bei komplexeren Schritten soll über dem tatsächlichen Inhalt des Schrittes eine kurze Beschreibung angezeigt werden, die den Nutzer über die Ziele des Schrittes informiert und gegebenenfalls Hinweise zur Eingabe der benötigten Daten gibt. Darunter befindet sich dann der eigentliche Inhalt des Schrittes, der je nach Schritt unterschiedlich gestaltet ist.

Schemaauswahl

Der Upload soll einfach gehalten werden und dem Nutzer nur die Möglichkeit geben, die beiden zu vergleichenden Schemata auszuwählen und hochzuladen. Da davon ausgegangen wird, dass die neue Version des Schemas mithilfe des RDF-Architects erstellt wurde, müssen hierfür der Name des Datensatzes und des Graphen angegeben werden, um das Schema in der Datenbank identifizieren zu können. Die alte Version wird hingegen als Datei hochgeladen.

Validierung der Klassenumbenennungen

Die Validierung der Klassenumbenennungen soll dem Nutzer sowohl eine Übersicht über die erkannten Klassenänderungen geben, als auch die Möglichkeit bieten, die vorgeschlagenen Umbenennungen anzupassen. Zu diesem Zweck muss eine Darstellung gefunden werden, die die benötigten Informationen und Eingabefelder übersichtlich anordnet.

Ein erster Entwurf sah vor, alle Klassenänderungen in einer Tabelle darzustellen (s. Abbildung 4.5), mit Spalten für den ursprünglichen Klassennamen, den vorgeschlagenen neuen Namen, den Typ der Änderung und die berechnete Ähnlichkeit im Fall einer Umbenennung. Allerdings bringt diese Darstellung auch einige Nachteile mit sich. Zum einen werden Klassenänderungen von verschiedenen Typen gemischt und nicht klar voneinander getrennt, was die Darstellung unübersichtlich macht und es dem Nutzer erschwert, bestimmte Änderungen zu finden. Zum anderen ist bei dieser Darstellung nicht klar, wie Umbenennungen am Besten bearbeitet werden können.

Old Name	New Name	Type	Confidence
-	AddedClass	Addition	-
DeletedClass	-	Deletion	-
ChangedClass	ChangedClass	Change	-
OldRenamedClass	NewRenamedClass	Rename	80%

Abbildung 4.5: Darstellung der Klassenänderungen in einer Tabelle

Eine Möglichkeit wäre, bei gelöschten und umbenannten Klassen die zweite Spalte als Dropdown-Menü darzustellen, in dem der Nutzer aus den hinzugefügten Klassen auswählen kann. So könnten erkannte Umbenennungen bearbeitet und aufgelöst

werden und gelöschte Klassen könnten mit neu hinzugefügten Klassen verknüpft werden. Allerdings müsste dann beim Ändern einer Löschung zu einer Umbenennung die bisher neue Klasse als eigene Zeile aus der Tabelle entfernt werden. Dieses ständige Rerendern der Tabelle wäre für den Nutzer verwirrend und erschwert die Bedienung.

Daher wurde sich entschieden, die verschiedenen Arten von Änderungen in drei separaten Bereichen darzustellen (s. Abbildung 4.6).

Der erste Bereich stellt alle gelöschten oder umbenannten Klassen in einer ähnlichen tabellarischen Form dar wie vorher. Dabei wird in der ersten Spalte der ursprüngliche Name der Klasse angezeigt, in der zweiten Spalte der vorgeschlagene neue Name, und in der dritten Spalte die berechnete Ähnlichkeit der beiden Klassen. Falls es sich nicht um eine Umbenennung handelt, sondern die Klasse gelöscht wurde, wird dementsprechend kein neuer Name in der zweiten Spalte angezeigt und auch die Ähnlichkeit bleibt leer.

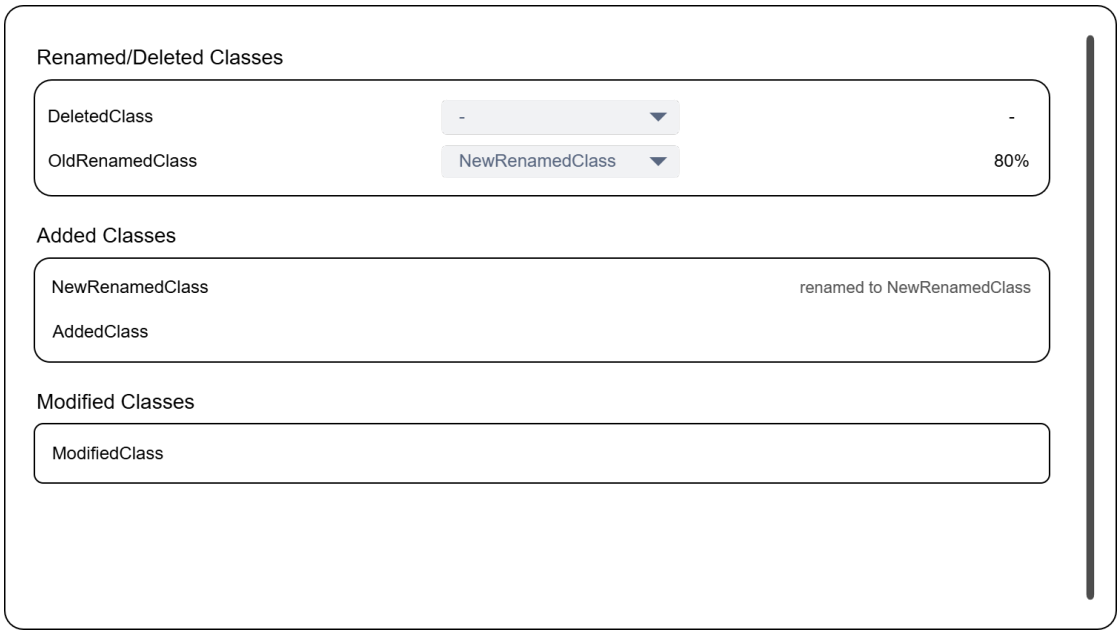


Abbildung 4.6: Darstellung der Klassenänderungen in drei Bereichen

Der zweite Bereich zeigt alle neu hinzugefügten Klassen als einfache Liste an. Allerdings wird neben vermeintlich neuen Klassen, die tatsächlich Teil einer Umbenennung sind, der alte Name der Klasse als Hinweis am rechten Rand der Zeile angezeigt. Zuletzt folgt im dritten Bereich eine Auflistung aller Klassen, die sich nur in ihren Eigenschaften geändert haben.

Diese Aufteilung in drei Bereiche soll dem Nutzer helfen, die verschiedenen Ar-

ten von Änderungen besser zu unterscheiden und fasst die Umbenennungen, die noch potentiell angepasst werden müssen, in einem eigenen Bereich zusammen. Zudem müssen keine Einträge dynamisch hinzugefügt oder entfernt werden, was die Bedienung vereinfacht.

Validierung der Property-Umbenennungen

Die Validierung der Property-Umbenennungen ist in drei Unterschritte aufgeteilt, entsprechend der drei Arten von Properties. Die Darstellung entspricht dabei der Darstellung der Klassenumbenennungen, mit dem Unterschied, dass Property-Umbenennungen, die zu einer Klasse gehören, in einem Bereich gruppiert sind (s. Abbildung 4.7). Dadurch lassen sich die Property-Umbenennungen besser im Kontext der jeweiligen Klasse betrachten, und es wird sofort ersichtlich, welche anderen Properties auf der Klasse geändert wurden. Da die wiederholte Anzeige der Änderungen pro Klasse jedoch sehr viel Platz einnimmt, lassen sich die einzelnen Klassenbereiche einklappen, woraufhin nur noch der Klassenname angezeigt wird.

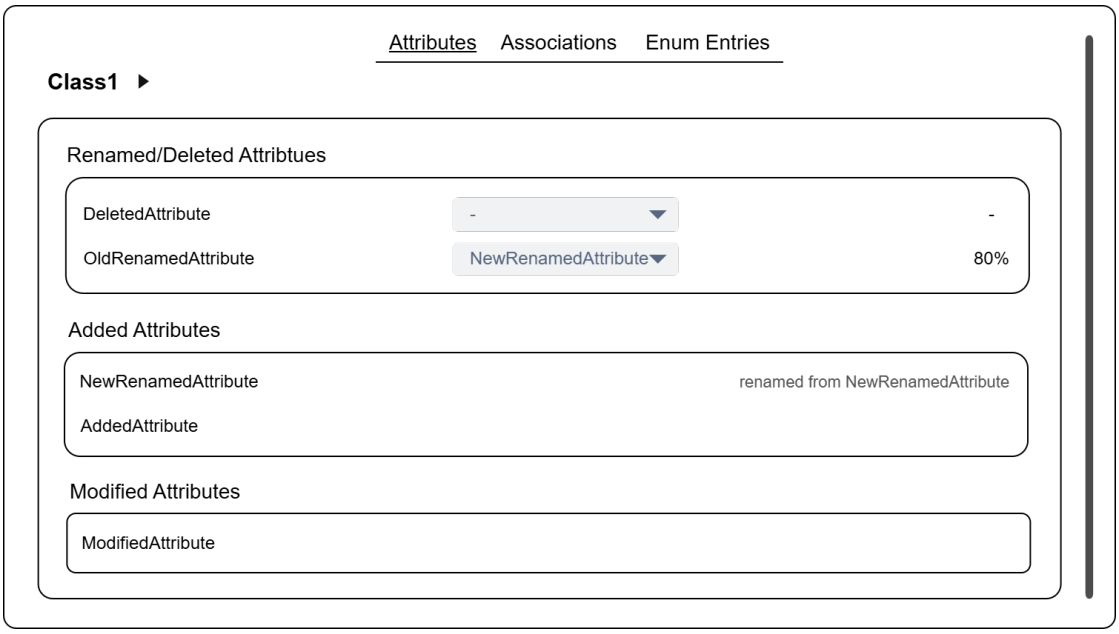


Abbildung 4.7: Darstellung der Property-Änderungen gruppiert nach Klasse

Eingabe der Default-Werte

Default-Werte sind wie auch die Property-Umbenennungen nach Art der Properties in Unterschritte aufgeteilt und innerhalb dieser Unterschritte pro Klasse gruppiert.

Abbildung 4.8 zeigt, wie die Eingabe der Default-Werte für Attribute dargestellt werden kann. Jede Klasse mit Attributen, die Default-Werte benötigen, zeigt diese in einer Tabelle an. Verpflichtende Attribute werden dabei mit einem roten Sternchen markiert.

Eine Zeile enthält immer den Namen des Attributs, den Typ der Änderung sowie ein Eingabefeld für den Default-Wert. Handelt es sich bei dem Datentyp des Attributs um ein Enum, wird statt des Eingabefelds ein Dropdown-Menü angezeigt, in dem der Nutzer einen der möglichen Enum Entries auswählen kann. Spezifisch bei optionalen Attributen hat der Nutzer zudem die Möglichkeit, auch dieses optionale Attribut auf den Klassen zu instanziieren.

Bei der Eingabe der SPARQL-Patterns für Assoziationen (vgl. Unterunterabschnitt 2.2.3) entfällt das „Init Optional“-Feld und statt der einzeiligen Eingabefelder werden mehrzeilige Textbereiche angezeigt, um die Eingabe der längeren Patterns zu erleichtern.

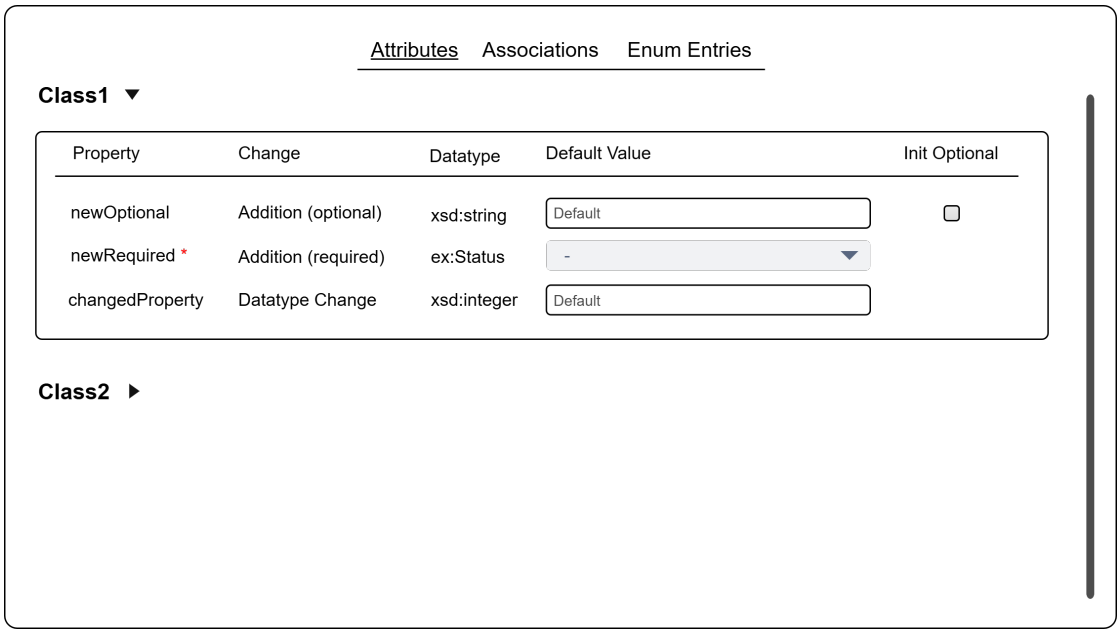


Abbildung 4.8: Eingabe der Default-Werte

Generieren des Skriptes

Der letzte Schritt des Wizards lässt den Nutzer mittels eines Buttons das Migrationsskript generieren und herunterladen. Dabei wird dem Nutzer eine kurze Anleitung zum weiteren Vorgehen angezeigt, die ihn darauf hinweist, die migrierten Daten mithilfe von SHACL zu validieren.

5 Implementierung

Dieses Kapitel beschreibt die Umsetzung des in Kapitel 4 entwickelten Ansatzes zur Schemamigration. Abschnitt 5.1 erläutert zunächst den Aufbau des Backends und welche Anpassungen für die Implementierung der Schemamigration notwendig waren. Anschließend werden in Abschnitt 5.2 die entwickelten Datenmodelle beschrieben, bevor in Abschnitt 5.3 bis Abschnitt 5.6 die Implementierung der einzelnen Schritte im Backend detailliert erläutert werden. Zum Schluss wird in Abschnitt 5.7 noch die Umsetzung des Migration-Wizards im Frontend vorgestellt.

5.1 Übersicht Backend

Entsprechend der hexagonalen Architektur ist die Implementierung des Backends in verschiedene Schichten unterteilt, um die verschiedenen Verantwortlichkeiten zu trennen. Abbildung 5.1 zeigt eine Übersicht über die verschiedenen Schichten und die wichtigsten Klassen, die in diesen Schichten angelegt wurden.

Die folgenden Abschnitte beschreiben die einzelnen Komponenten im Detail.

5.1.1 REST-Schnittstelle

Für jeden Migrationsschritt wurden entsprechende Endpunkte angelegt, jeweils zum Fetchen des Datenstandes vor der Nutzereingabe, sowie zum Absenden der Eingaben und Berechnung des Ergebnisses. Diese Endpunkte sind auf verschiedene Controller-Klassen verteilt, die jeweils für einen Migrationsschritt zuständig sind. Die Controller sind dabei wie im Rest des Projekts schlank gehalten und beinhalten nur die Logik, die spezifisch für die Kommunikation über HTTP ist. Die eigentliche Geschäftslogik ist in Service-Klassen ausgelagert, die von den Controllern über Interfaces aufgerufen werden.

Die implementierten Controller sind im Folgenden aufgelistet:

- MigrationContextRESTController: Enthält einen Endpunkt zum Starten des Migrationsprozesses, der zwei Schema-Versionen entgegennimmt und den Vergleich der beiden Versionen anstößt. Enthält einen weiteren Endpunkt zum Zurücksetzten der Migrations-Session.
- ClassRenamingRESTController: Enthält Endpunkte zum Fetchen der erkannten Klassenumbenennungen und zum Absenden der vom Nutzer validierten

Klassenumbenennungen.

- PropertyRenamingRESTController: Enthält Endpunkte zum Fetchen der erkannten Property-Umbenennungen und zum Absenden der vom Nutzer validierten Property-Umbenennungen.
- DefaultValuesRESTController: Enthält Endpunkte zum Fetchen der relevanten Informationen für das Setzen von Default-Werten und zum Absenden der konfigurierten Default-Werte.
- MigrationScriptRESTController: Enthält einen Endpunkt zum Generieren des finalen Migrations-Skripts.

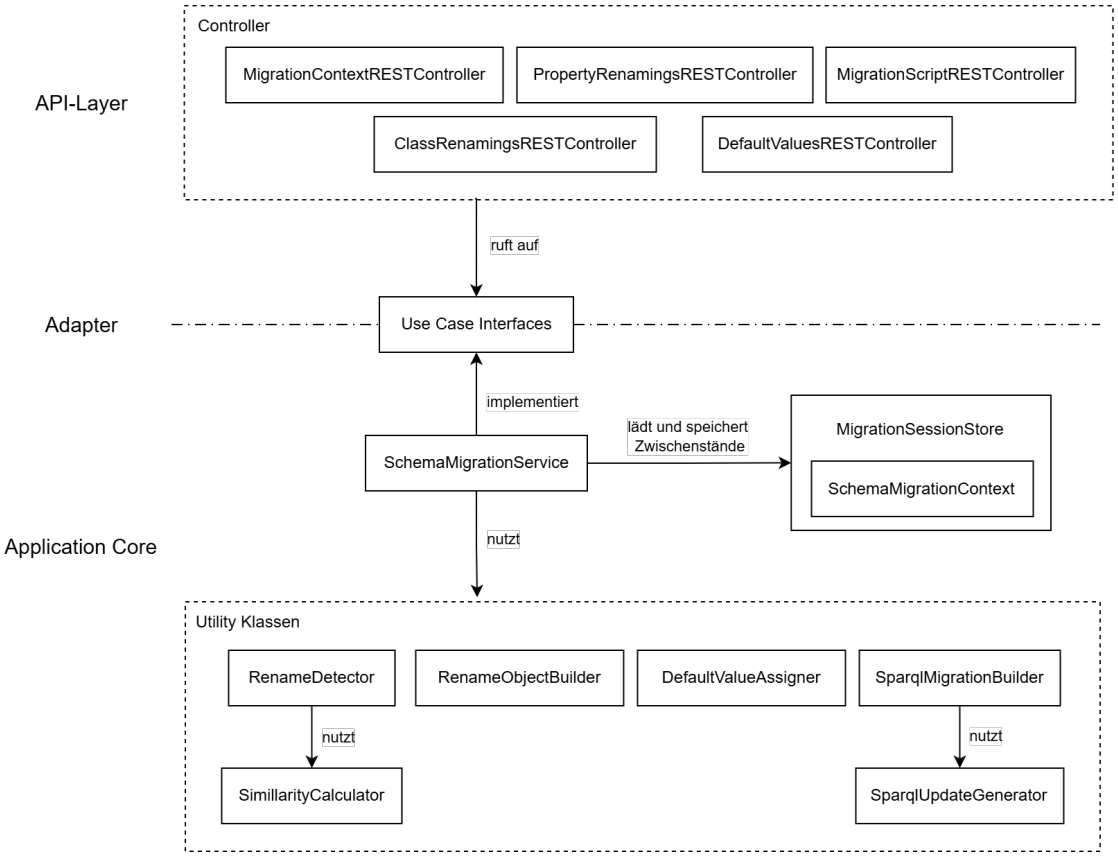


Abbildung 5.1: Übersicht über die Backend-Implementierung

5.1.2 Geschäftslogik

Für jeden der erstellten Endpunkte wurde ein entsprechendes Interface angelegt, das als Adapter-Layer zwischen den REST-Controllern und der eigentlichen Ge-

schäftslogik dient. Die genauen Namen dieser Interfaces sind aufgrund der Anzahl der Interfaces in Abbildung 5.1 nicht dargestellt. Die Interfaces werden alle von dem neu angelegten `SchemaMigrationService` implementiert, der dadurch eine Methode pro Endpunkt bereitstellt. Da die gesamte Logik der Schemamigration allerdings zu umfangreich ist, um sie in einer einzigen Klasse zu verwalten, wurden für einige Migrationsschritte weitere Utility-Klassen erstellt, z.B. für die Erkennung der Umbenennungen oder dem Generieren des SPARQL-Skriptes. Die genaue Umsetzung der Migrationsschritte und der genutzten Datenmodelle wird in Abschnitt 5.2 bis Abschnitt 5.6 beschrieben.

Migrations-Kontext

Eine Anforderung an den Migrations-Wizard ist, dass der Nutzer zu vorherigen Schritten des Wizards zurückkehren und seine Eingaben anpassen kann. Um dies zu ermöglichen, müssen die Ergebnisse der einzelnen Schritte zwischengespeichert werden. Zu diesem Zweck wurden die beiden Klassen `SchemaMigrationContext` und `MigrationSessionStore` erstellt. Der `SchemaMigrationContext` speichert dabei die ursprünglichen Eingaben und Ergebnisse der einzelnen Migrationsschritte, während der `MigrationSessionStore` die verschiedenen Kontexte für die einzelnen Sessions verwaltet.

Der `SchemaMigrationService` nutzt diesen Store um vor jeder Anfrage aus dem Frontend die gespeicherten Daten des vorherigen Schrittes zu laden. Im Fall von Get-Anfragen werden daraufhin die benötigten Informationen wie Umbenennungen aus dem letzten Stand extrahiert und an das Frontend geliefert. Im Fall von Post-Anfragen wird der letzte Stand anhand der neuen Eingaben aktualisiert und in einer neuen Variable im Store gespeichert.

Durch die Speicherung des Kontexts sind die Migrations-Endpunkte zwar nicht mehr strikt zustandslos, jedoch wird die Geschäftslogik des Migration-Services dadurch deutlich vereinfacht.

5.2 Datenmodelle

Für die Repräsentation der abstrakten Änderungen an Ressourcen, hier auch semantische Änderungen genannt, mussten im Backend des RDF-Architects einige neue Klassen zur Speicherung der Daten angelegt werden.

5.2.1 Bestehende Datenmodelle

Im Rahmen des Schemavergleichs, der bereits vor dieser Arbeit implementiert war, wurden einige Datenmodelle zur Repräsentation von Änderungen an Ressourcen erstellt. Als Basis dient hier die Klasse `TripleResourceChange`, die eine IRI, ein Label und eine Liste von `TriplePropertyChange`-Objekten enthält. `TriplePropertyChange` repräsentiert dabei eine Änderung an einem einzelnen Tripel einer Ressource und enthält die IRI des Prädikates, sowie den alten und neuen Wert des Objekts.

Während Properties wie Attribute, Assoziationen und Enum-Entries direkt als `TripleResourceChange`-Objekte dargestellt werden können, wurden für Klassen und Packages spezielle Unterklassen erstellt, welche jeweils noch eine Liste an den zugehörigen Properties bzw. Klassen enthalten. Dadurch entsteht eine hierarchische Struktur, die die Verschachtlung von Packages, Klassen und deren Properties übersichtlich darstellt.

5.2.2 SemanticResourceChange

Um abstraktere Änderungen an Ressourcen darstellen zu können, wie beispielweise Umbenennungen, reichen die bestehenden Datenmodelle für Tripel-Änderungen nicht aus. Bei der Erstellung neuer Datenmodelle für die Darstellung von semantischen Änderungen bietet es sich jedoch an, sich an den bestehenden Modellen zu orientieren und eine ähnliche hierarchische Struktur und Polymorphie zu verwenden. Dafür wurde zunächst die Basis-Klasse `SemanticResourceChange` erstellt (s. Code 5.1).

Wie auch bei der `TripleResourceChange`-Klasse werden für die Ressourcen eine IRI, ein Label und eine Liste von Änderungen gespeichert. Zusätzlich zu diesen Informationen speichert die Klasse jedoch auch einen `SemanticResourceChangeType`, der eine abstrakte Art der Änderung angibt, sowie eine alte IRI, die bei Umbenennungen verwendet wird.

Die verschiedenen Arten von Änderungen, die durch `SemanticResourceChangeType` definiert werden, sind:

- `ADD`,
- `ADDED_FROM_INHERITANCE`,
- `DELETE`,
- `DELETED_FROM_INHERITANCE`,
- `CHANGE`,

- `RENAME`,

Neben den erwartbaren Änderungsarten wie `ADD`, `DELETE` und `CHANGE` gibt es auch die Änderungsarten `ADDED_FROM_INHERITANCE` und `DELETED_FROM_INHERITANCE`. Die Relevanz dieser Änderungsarten wird in Abschnitt 5.5 erläutert.

```
@Data
@EqualsAndHashCode(callSuper=false)
@SuperBuilder
@AllArgsConstructor
@NoArgsConstructor
@JsonTypeInfo(...)
@JsonSubTypes({...})
public sealed class SemanticResourceChange permits ... {
    protected SemanticResourceChangeType semanticResourceChangeType;
    protected String label;
    protected String oldIRI;
    protected String iri;

    @Builder.Default
    protected List<SemanticFieldChange> changes = new ArrayList<>();

    // Konstruktoren ...
}
```

Code 5.1: SemanticResourceChange

5.2.3 SemanticFieldChange

Ähnlich wie bei `TriplePropertyChange` wird auch die `SemanticFieldChange`-Klasse verwendet, um Änderungen an einzelnen Feldern einer Ressource zu repräsentieren (s. Code 5.2). Die IRI des geänderten Prädikats wird dabei durch einen `SemanticFieldChangeType` ersetzt, der die Art der Änderungen und deren Relevanz für die Instanzdaten beschreibt. Bei der Änderung der Multiplicity eines Attributes von `M:0..1` zu `M:1..1` kann so beispielweise direkt angegeben werden, dass die Änderung das Attribut zu einem Pflichtfeld macht. Diese Angabe des Typs reduziert das wiederholte Parsen und Auswerten der alten und neuen Werte der Felder. Auch das Entfernen des `concrete`-Stereotyps kann von den anderen Stereotypänderungen unterschieden und als `MADE_ABSTRACT` angegeben werden.

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class SemanticFieldChange {
    private SemanticFieldChangeType semanticFieldChangeType;
    private String from;
    private String to;

    // Konstruktoren ...
}
```

Code 5.2: SemanticFieldChange

5.2.4 Unterklassen von SemanticResourceChange

Um die benötigten Informationen für die Migration der verschiedenen Ressourcentypen zu speichern, wurde für jede Art von Ressource eine Unterklasse von `SemanticResourceChange` erstellt. `SemanticClassChange` speichert dabei zusätzlich drei verschiedene Listen für die unterschiedlichen Properties der Klasse (Attribute, Assoziationen, Enum-Entries), sowie drei weitere Listen, um die potentiellen Umbenennungen dieser Properties zu speichern.

Code 5.3 zeigt die zusätzlichen Felder der `SemanticAttributeChange`-Klasse, die besonders für die Zuweisung von Default-Werten benötigt werden. Die Klasse speichert dabei den Datentyp des Attributes, da dieser nicht zwingend in der Liste der Änderungen enthalten sein muss, und den primitiven XSD-Datentyp. Dieser primitive Datentyp wird, wie in Unterabschnitt 2.2.4 beschrieben, bei Attributen mit selbst definierten Datentypen benötigt, um den Default-Wert in dem RDF-Graphen korrekt zu typisieren.

Zudem speichert die Klasse den vom Benutzer angegebenen Default-Wert, ob das Attribut optional ist und ob der Default-Wert auch für optionale Attribute instanziiert werden soll. Abschließend wird im Fall von Enum-Attributen noch die Liste der erlaubten Enum-Entries gespeichert.

`SemanticAssociationChange` arbeitet sehr ähnlich zu `SemanticAttributeChange`. Statt Datentyp und primitivem Datentyp wird hier die Range gespeichert, und statt dem Default-Wert das vom Benutzer angegebene SPARQL-Mapping. Analog zum `optional` Wert bei Attributen wird auch hier der `associationUsed`-Wert gespeichert, der angibt, ob die Assoziation instanziiert werden soll (s. Code 5.4).

```java
@Data
@SuperBuilder
@EqualsAndHashCode(callSuper = true)
@NoArgsConstructor
public final class SemanticAttributeChange extends
    SemanticResourceChange {
    private String dataType;
    private String primitiveDataType;
    private String defaultValue;
    private boolean optional;
    private boolean forceDefaultValue = false;
    @Builder.Default
    private List<String> allowedValues = new ArrayList<>();

    // Konstruktoren ...
}
```

Code 5.3: SemanticAttributeChange

```java
@Data
@SuperBuilder
@NoArgsConstructor
@EqualsAndHashCode(callSuper = true)
public final class SemanticAssociationChange extends
    SemanticResourceChange {
    private String range;
    private String mapping;
    private boolean associationUsed;

    // Konstruktoren ...
}
```

Code 5.4: SemanticAssociationChange

`SemanticEnumEntryChange` speichert lediglich eine Liste an erlaubten Werten für den Enum-Entry und den vom Benutzer angegebenen Ersatzwert, falls der Enum-Entry gelöscht wurde (s. Code 5.5).

Für Packages wurde keine eigene Unterklasse erstellt, da Packages keine Relevanz für die Instanzdaten haben. Die Package-Ebene wird in der Klassenhierar-

```java
@Data
@SuperBuilder
@NoArgsConstructor
@EqualsAndHashCode(callSuper = true)
public final class SemanticEnumEntryChange extends
    SemanticResourceChange {
    private String replacementValue;
    @Builder.Default
    private List<String> allowedValues = new ArrayList<>();

    // Konstruktoren ...
}
```

Code 5.5: SemanticEnumEntryChange

chie daher nicht dargestellt und semantische Änderungen werden als Liste von `SemanticClassChange`-Objekten gespeichert.

5.2.5 Umbenennungen

Umbenennungen von Ressourcen können, wie gelöschte und hinzugefügte Klassen, als semantische Änderungen dargestellt werden, diesmal mit dem `SemanticResourceChangeType.RENAME` Typ. Da diese Umbenennungen allerdings zuerst von dem Benutzer validiert werden müssen, müssen vom Backend auch die potentiellen Umbenennungspaare gespeichert werden. Dafür wurde eine eigene Klasse `RenameCandidate` angelegt, die jeweils Referenzen der alten und neuen Ressource sowie einen errechneten Ähnlichkeitswert der beiden Ressourcen speichert (s. Code 5.6).

Dabei handelt es sich um eine generische Klasse, die als Typ alle Klassen akzeptiert, die von `SemanticResourceChange` erben. Dadurch lässt sich die Klasse nutzen, um Umbenennungen für alle möglichen Ressourcentypen darzustellen, stellt aber gleichzeitig sicher, dass die alte und neue Ressource den selben Typ `T` haben. Ein weiterer Vorteil des generischen Typs zeigt sich zudem, wenn Listen an Umbenennungkandidaten gespeichert werden, da eine Liste Liste von Klassenumbenennungen mithilfe der Typangabe auch nur `RenameCandidate<SemanticClassChange>` Objekte enthalten kann.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class RenameCandidate <T extends SemanticResourceChange> {
    private T oldResource;
    private T newResource;
    private double confidenceScore;

    // Konstruktoren ...
}
```

Code 5.6: RenameCandidate

5.3 Semantische Änderungen analysieren

Wie bereits die Datenmodelle kann auch das Erkennen der semantischen Änderungen auf dem bestehenden Schemavergleich aufbauen. Der Schemavergleich berechnet die Unterschiede zwischen zwei Schemata auf Tripel-Ebene und speichert die Ergebnisse in einer Liste von `TriplePackageChange`-Objekten. Dieses Ergebnis kann als Preprocessing Schritt betrachtet und von der Extraktion der semantischen Änderungen weiterverwendet werden.

Die Logik für diese Transformation befindet sich in der `SemanticChangeAnalyser`-Klasse. Die einzige öffentliche Methode dieser Klasse ist `getSemanticChanges`, die eine Liste von `TriplePackageChange`-Objekten entgegennimmt und eine Liste von `SemanticClassChange`-Objekten zurückgibt. Diese Methode ruft intern weitere private Methoden auf, die jeweils für eine bestimmte Art von Ressource zuständig sind und die semantischen Änderungen für diese Ressource auslesen.

Code 5.7 zeigt beispielhaft, wie die semantischen Änderungen von Attributen erkannt werden. Die Methode legt ein neues Objekt der Klasse `SemanticAttributeChange` an und setzt deren Typ mittels der Hilfsmethode `getResourceChangeType`. Anschließend werden die einzelnen Tripel-Änderungen durchlaufen und in `SemanticFieldChange`-Objekte umgewandelt. Hier kommt ebenfalls eine Hilfsmethode zum Einsatz, die das Prädikat der Tripel-Änderung auf einen semantischen Änderungstyp mappt. Abschließend wird das neue Change-Objekt an die aufrufende Methode zurückgegeben, welche die Ergebnisse aggregiert. Die Methoden für Klassen und Packages würden zudem noch weitere Methoden für ihre Unterressourcen aufrufen.

```
private SemanticAttributeChange getSemanticChangesForAttribute(
    TripleResourceChange attribute) {
    var semanticChangeObject = new SemanticAttributeChange(attribute);
    var attributeChanges = attribute.getChanges();

    semanticChangeObject.setSemanticResourceChangeType(
        getResourceChangeType(attributeChanges));

    var changes = semanticChangeObject.getChanges();
    for (var propertyChange : attributeChanges) {
        var action = new SemanticFieldChange(propertyChange);
        var mappedType =
            SemanticFieldChangeTypeMapper.mapPredicateToChangeType("
            Attribute", propertyChange);
        if (mappedType != null) {
            action.setSemanticFieldChangeType(mappedType);
            changes.add(action);
        }
    }

    return semanticChangeObject;
}
```

Code 5.7: Erkennen semantischer Änderungen von Attributen

5.3.1 FieldChangeType-Mapper

Der `SemanticFieldChangeTypeMapper` wird verwendet, um die IRIs der Prädikate auf die entsprechenden semantischen Änderungstypen zu mappen. Einige der Prädikate können dabei für jede Ressource gleich behandelt werden, wie z.B. `rdfs:label` oder `rdfs:comment`. Diese Prädikate werden von der generellen Methode `mapCommonPredicatesToChangeType` behandelt. Andere Prädikate treten jedoch nur auf bestimmten Ressourcentypen auf oder werden sogar auf verschiedenen Arten von Ressourcen unterschiedlich interpretiert. Datatype und Range Änderungen werden auf Attributen beispielsweise zu `DATATYPE_CHANGED` zusammengefasst, während eine Range Änderungen auf Assoziation als `TARGET_CHANGED` interpretiert wird. Um diese Fälle behandeln zu können, bekommt die `mapPredicateToChangeType`-Methode zusätzlich den Ressourcentyp als Parameter übergeben. Anhand dieses Typs können dann eigene Mapping-Methoden für die verschiedenen Ressourcentypen aufgerufen werden.

5.3.2 getRessourceChangeType

`getRessourceChangeType` ist eine Hilfsmethode, die den Änderungstyp einer Ressource bestimmt (s. Code 5.8). Hierfür wird die Liste der Änderungen nach einem `TriplePropertyChange`-Objekt mit dem Prädikat `rdf:type` durchsucht. Wie in Unterabschnitt 2.2.3 erläutert, besitzt jeder Ressourcentyp zwingend ein Tripel mit diesem Prädikat, weshalb eine Änderung dieses Tripels von einem Null-Wert zu einem konkreten Objekt als Beweis genutzt werden kann, dass die Ressource neu hinzugefügt wurde. Umgekehrt bedeutet eine Änderung von einem konkreten Objekt zu einem Null-Wert, dass die Ressource gelöscht wurde. Falls keine Änderung des `rdf:type`-Prädikats gefunden wird, wird der Änderungstyp `CHANGE` zurückgegeben, da sich in diesem Fall nur andere Eigenschaften der Ressource geändert haben.

```
private SemanticResourceChangeType getResourceChangeType(List<
    TriplePropertyChange> changes) {
    var typeChange = changes.stream()
                          .filter(c -> c.getPredicate().equals(
                              RDF.type.toString()))
                          .findFirst()
                          .orElse(null);
    if (typeChange != null) {
        if (typeChange.getFrom() == null) {
            return SemanticResourceChangeType.ADD;
        } else if (typeChange.getTo() == null) {
            return SemanticResourceChangeType.DELETE;
        }
    } else {
        return SemanticResourceChangeType.CHANGE;
    }
    return null;
}
```

Code 5.8: Erkennen semantischer Änderungen von Attributen

5.4 Umbenennungserkennung

Die Implementierung der Umbenennungserkennung ist auf mehrere verschiedene Utility-Klassen aufgeteilt: den `RenameDetector`, der die Hauptlogik der Erkennung enthält, den `SimilarityCalculator`, der die Ähnlichkeit zweier Ressourcen berechnet, und den `RenameObjectBuilder`, der für die bestätigten Umbenennungen neue Change-Objekte erstellt.

5.4.1 RenameDetector

Der `RenameDetector` wird von dem `SchemaMigrationService` aufgerufen, um sowohl Klassen- als auch Property-Umbenennungen zu erkennen. Dafür stellt er die beiden öffentlichen Methoden `detectClassRenames` und `detectPropertyRenames` zur Verfügung. Beide Methoden arbeiten ähnlich, indem sie jeweils eine Liste an Ressourcen übergeben bekommen, die auf Umbenennungen überprüft werden sollen. Im Fall von Klassen handelt es sich dabei um die `SemanticClassChange`-Objekte, die im vorherigen Schritt erstellt wurden, und im Fall von Properties um eine generische Liste an `SemanticRessourceChange`-Objekten, die die Attribute, Assoziationen oder Enum Entries einer Klasse enthalten kann. Beide Methoden iterieren über die übergebenen Ressourcen, um anhand des jeweiligen `SemanticResourceChangeType` separate Listen für hinzugefügte und gelöschte Ressourcen zu erstellen. Anschließend rufen sie die private Methode `matchBySimilarity` auf, die anhand dieser Listen potentielle Umbenennungen erkennt.

Code 5.9 zeigt die Implementierung von `matchBySimilarity`. Die Methode erstellt dafür zunächst eine Kopie der hinzugefügten Ressourcen als Set, um später erkannte Umbenennungen daraus entfernen zu können. Anschließend wird über die Liste der gelöschten Ressourcen iteriert und jeweils mit allen noch nicht zugeordneten hinzugefügten Ressourcen die Ähnlichkeit berechnet. Hierfür wird der `SimilarityCalculator` verwendet, der einen Ähnlichkeitswert zwischen 0 und 1 zurückgibt. Nachdem alle Vergleiche durchgeführt wurden, wird überprüft, ob der höchste Ähnlichkeitswert über einem definierten Schwellwert liegt (vgl. Unterabschnitt 4.2.5). Dieser Schwellwert ist in der Klasse als Konstante definiert, um ihn bei Bedarf einfach anpassen zu können.

Falls der höchste Ähnlichkeitswert über dem Schwellwert liegt, wird aus den beiden Ressourcen ein `RenameObject` erstellt und zu der Ergebnisliste hinzugefügt. Die hinzugefügte Ressource wird zudem aus dem Set entfernt, um Mehrfachzuordnungen zu vermeiden.

```
private <T extends SemanticResourceChange> List<RenameCandidate<T>>
    matchBySimilarity(List<T> added, List<T> deleted) {
    List<RenameCandidate<T>> renames = new ArrayList<>();
    Set<T> unmatchedAdded = new HashSet<>(added);

    for (T deletedItem : deleted) {
        double bestScore = 0.0;
        T bestMatch = null;

        for (T newItem : unmatchedAdded) {
            double score = SimilarityCalculator.calculateSimilarity(
                newItem, deletedItem);
            if (score > bestScore) {
                bestScore = score;
                bestMatch = newItem;
            }
        }

        if (bestMatch != null && bestScore > SIMILARITY_THRESHOLD) {
            unmatchedAdded.remove(bestMatch);
            renames.add(new RenameCandidate<>(deletedItem, bestMatch,
                bestScore));
        }
    }
    return renames;
}
```

Code 5.9: Erkennung von Umbenennungen mittels Ähnlichkeitsvergleich

5.4.2 SimilarityCalculator

Der `SimilarityCalculator` wird vom `RenameDetector` verwendet, um die Ähnlichkeit zweier Ressourcen zu berechnen. Die Berechnung der Ähnlichkeit findet dabei anhand des in Abschnitt 4.2 entwickelten Verfahrens statt.

Die öffentliche Methode `calculateSimilarity` nimmt ein generisches gelöschtes und hinzugefügtes `SemanticRessourceChange`-Objekt entgegen. Zuerst wird die lexikalische Ähnlichkeit der Label der beiden Ressourcen mithilfe der privaten Methode `calculateLabelSimilarity` berechnet. Diese Methode erstellt Bigram-Vektoren der beiden Label und nutzt dann die von Apache implementierte Kosinus-Ähnlichkeit, um den lexikalischen Ähnlichkeitswert zu berechnen.

Daraufhin werden je nach Ressourcentyp weitere private Methoden aufgerufen, die die strukturelle Ähnlichkeit der beiden Ressourcen berechnen. Einfache Felder können dabei mithilfe der `compareFieldValues`-Methode verglichen werden, die anhand eines übergebenen `SemanticFieldChangeType` die relevanten Felder extrahiert und deren Werte vergleicht. Stereotypen und Properties einer Klasse müssen hingegen speziell behandelt werden, da es sich hierbei um Listen handelt. Die Methoden `calculateStereotypeSimilarity` und `calculatePropertySimilarity` berechnen daher die Ähnlichkeit dieser Listen, indem sie die Anzahl der übereinstimmenden Einträge durch die Gesamtanzahl der Einträge teilen. Code 5.10 zeigt die Implementierung der `calculateStereotypeSimilarity`-Methode.

```
private double calculateStereotypeSimilarity(SemanticClassChange added,
    SemanticClassChange deleted) {
    var deletedStereotypes = deleted.getChanges().stream()
            .filter(c -> c.getSemanticFieldChangeType() ==
                SemanticFieldChangeType.STEREOTYPE_REMOVED)
            .map(SemanticFieldChange::getFrom)
            .toList();

    var addedStereotypes = added.getChanges().stream()
            .filter(c -> c.getSemanticFieldChangeType() ==
                SemanticFieldChangeType.STEREOTYPE_ADDED)
            .map(SemanticFieldChange::getTo)
            .toList();

    if (deletedStereotypes.isEmpty()) {
        return 1.0;
    }

    var matchingStereotypes = deletedStereotypes.stream()
            .filter(addedStereotypes::contains)
            .count();

    return (double) matchingStereotypes / deletedStereotypes.size();
}
```

Code 5.10: Berechnung der Stereotyp-Ähnlichkeit

Nachdem sowohl die lexikalische als auch die strukturelle Ähnlichkeit berechnet wurden, führt die `calculateSimilarity`-Methode diese beiden Werte zu einem Gesamtergebnis zusammen und gibt dieses zurück. Die Werte für die Gewichtung der struk-

turellen und lexikalischen Ähnlichkeit sind wie bereits der `SIMILARITY_THRESHOLD` in der Klasse als Konstanten definiert, um sie leicht anpassen zu können.

5.4.3 Speichern der bestätigten Umbenennungen

Nachdem die erkannten Umbenennungen vom Nutzer im Frontend bestätigt bzw. korrigiert wurden, müssen diese in dem `SchemaMigrationContext` aufgenommen werden. Um die validierten Klassenumbenennungen zu speichern, erstellt die `confirmClassRenamings`-Methode des `SchemaMigrationService` eine Kopie der bisherigen semantischen Änderungen und iteriert dann über die Liste an Klassenumbenennungen (s. Code 5.11). Für jede Umbenennung werden die alten und neuen `SemanticClassChange`-Objekte aus der Kopie gelöscht und es wird ein neues `SemanticClassChange`-Objekt mithilfe des `RenameObjectBuilder` erstellt. Dieses neue Objekt stellt die zusammengeführten Änderungen der beiden ursprünglichen Objekte dar, wobei der Änderungstyp auf `RENAME` gesetzt wird.

```
@Override
    public void confirmClassRenamings(List<RenameCandidate<
        SemanticClassChange>> renames) {
        var context = migrationSessionStore.getContext();
        context.setRenameCandidates(renames);
        var classChanges = new ArrayList<>(context.getSemanticDiff());

        for (var rename : renames) {
            classChanges.remove(rename.getNewResource());
            classChanges.remove(rename.getOldResource());
            classChanges.add((SemanticClassChange)
                RenameObjectBuilder.createRenameObject(rename));
        }
        migrationSessionStore.getContext().setDiffAfterClassConfirm(
            classChanges);
    }
```

Code 5.11: Speichern der bestätigten Klassenumbenennungen

Die Methode zum Speichern der Property-Umbenennungen funktioniert ähnlich, muss jedoch zusätzlich über die Klassen iterieren, da die Properties in den `SemanticClassChange`-Objekten eingebettet sind. Danach wird analog zur Klassenumbenennung über die verschiedenen Listen der Properties iteriert, um die bestätigten Umbenennungen zu speichern.

5.4.4 RenameObjectBuilder

Der `RenameObjectBuilder` wird verwendet, um aus zwei `SemanticResourceChange`-Objekten ein neues Objekt zu erstellen, das die Umbenennung repräsentiert. Die öffentliche Methode `createRenameObject` nimmt dafür zwei generische `SemanticResourceChange`-Objekte entgegen und erstellt eine Kopie des hinzugefügten Objekts. Anschließend wird der Änderungstyp auf `RENAME` gesetzt und die alte IRI auf die IRI des gelöschten Objekts gesetzt. Daraufhin wird die Methode `mergeChanges` aufgerufen, die die Änderungen der Eigenschaften der beiden Objekte zusammenführt (s. Code 5.12).

```
public List<SemanticFieldChange> mergeChanges(List<SemanticFieldChange>
    added, List<SemanticFieldChange> deleted) {
    var result = new ArrayList<SemanticFieldChange>();
    var remainingAdded = new ArrayList<>(added);

    for (var deletedChange : deleted) {
        var addedChange = findMatchingChange(remainingAdded,
            deletedChange);

        if (addedChange != null) {
            processMergedChange(result, addedChange, deletedChange,
                remainingAdded);
        } else {
            result.add(deletedChange);
        }
    }

    result.addAll(remainingAdded);
    return result;
}
```

Code 5.12: Zusammenführen der Änderungen bei Umbenennungen

Dabei wird über die Änderungen der gelöschten Ressource iteriert und für jede Änderung nach einer passenden Änderung in der hinzugefügten Ressource gesucht. Falls eine passende Änderung gefunden wird, wird die Methode `processMergedChange` aufgerufen, die ein neues `SemanticFieldChange`-Objekt mit den entsprechenden alten und neuen Werten erstellt. Diese Methode filtert zudem Änderungen heraus, die nach dem Zusammenführen den gleichen alten und neuen Wert besitzen, da diese keine tatsächliche Änderung darstellen. Falls keine passende Änderung gefunden wird, wird die Änderung der gelöschten Ressource direkt zur Ergebnisliste

hinzugefügt. Abschließend werden alle noch nicht verarbeiteten Änderungen der hinzugefügten Ressource zur Ergebnisliste hinzugefügt und die Liste zurückgegeben.

Falls es sich bei der umbenannten Ressource um ein Property handelt, ist die Erstellung des neuen `SemanticRessourceChange`-Objekts an diesem Punkt abgeschlossen und das Objekt kann zurückgegeben werden. Handelt es sich allerdings um eine Klasse, müssen neben den Änderungen der Klasse selbst auch noch die Properties der Klasse zusammengeführt werden. Das Vorgehen hierfür ist ähnlich wie bei den Änderungen der Eigenschaften: Es wird über die verschiedenen Property-Listen der gelöschten Klasse iteriert und für jede Property nach einer passenden Property in der hinzugefügten Klasse gesucht. Falls eine passende Property gefunden wird, wird ein neues Change-Object erstellt und die Eigenschaften der Properties werden zusammengeführt. Anders als bei der umbenannten Ressource wird der `SemanticResourceChangeType` hierbei auf `CHANGE` gesetzt. Abschließend muss zudem die bisher erkannte `DOMAIN_CHANGE`-Änderung in eine `DOMAIN_RENAME`-Änderung geändert werden, damit die Instanzdaten bei der Migration beibehalten werden.

5.5 Default Werte

Damit neue verpflichtende Attribute und Assoziationen während der Migration initialisiert werden können, müssen dafür von Benutzer jeweils Default-Werte angegeben. Damit der Nutzer diese Angaben im Frontend machen kann werden allerdings einige Informationen benötigt, wie z.B. der Datentyp des neuen Attributs. Diese Informationen müssen vom Backend vor der Eingabe aus dem Schema extrahiert und den entsprechenden Change-Objekten hinzugefügt werden. In diesem Kapitel wird beschrieben, wie die benötigten Informationen für die verschiedenen Arten von Änderungen ermittelt und den Change-Objekten hinzugefügt werden.

5.5.1 Änderungen durch Vererbung

Bevor die Informationen zu den Default-Werten ermittelt werden können, stellt sich die Frage, welche Änderungen überhaupt Default-Werte benötigen. In Abschnitt 4.1 wurden bereits ein Großteil der Fälle beschrieben, welche sich wie folgt zusammenfassen lassen:

- Neue verpflichtende Attributen
- Neue optionale Attribute, falls der Nutzer diese instanziieren möchte
- Attribute, die von optional auf verpflichtend geändert wurden

- Neue Assoziationen mit einer minimalen Multiplizität von mindestens 1
- Assoziationen, deren minimale Multiplizität erhöht wurde
- Gelöschte Enum Entries, die einen potentiellen Ersatz brauchen

Alle diese Änderungen lassen sich direkt an einer bestimmten Änderung einer Eigenschaft oder, im Falle von Enum Entries, dem Löschen der gesamten Ressource festmachen. Es gibt jedoch auch eine weitere Änderungen, die indirekte Auswirkungen auf die Properties einer Klasse hat: Nämlich das Ändern der Superklasse.

Wird die Superklasse einer Klasse geändert oder gelöscht, müssen alle Properties, die bisher von der Superklasse und deren Superklassen geerbt wurden, von allen Instanzen der geänderten Klasse und deren erbenden Klassen entfernt werden. Ebenso müssen neu geerbte Properties auf diesen Instanzen hinzugefügt werden.

Um dieses Verhalten in der Migration behandeln und den neu geerbten Properties ebenfalls Default-Werte zuweisen zu können, wurde entschieden, diese Änderungen durch Vererbung ebenfalls mittels Change-Objekten auf den Properties abzubilden. Um diese neuen Change-Objekte zu erzeugen, wurde die Utility-Klasse `InheritanceChangeHandler` angelegt.

Da in den meisten Fällen nicht alle Klassen und ihre Properties einer Vererbungskette als Change-Objekte vorliegen, müssen diese Informationen separat aus den Schemata ausgelesen werden. Hierfür können einige bereits implementierte Hilfsfunktionen aus dem `CIMUtils`-Package genutzt werden. Um mit diesen Funktionen arbeiten zu können, müssen das alte und neue Schema zunächst in Form von Apache Jenas `Model`-Objekten vorliegen.

Der `InheritanceChangeHandler` besitzt eine öffentliche Methode `processInheritanceChanges`, die von dem `SchemaMigrationService` aufgerufen wird. Diese Methode erzeugt zunächst eine Map, die jeweils die bereits vorliegenden `SemanticClassChange`-Objekte auf ihre IRI abbildet und im Folgenden beim Erstellen der neuen Change-Objekte der Properties benötigt wird. Daraufhin iteriert die Methode über alle Klassenänderungen und prüft, ob eine Änderung der Superklasse vorliegt. Für jede Klasse, bei der eine solche Änderung vorliegt, wird dann die Methode `addPropertyChangesFromInheritance` aufgerufen.

Um die richtigen Properties zu identifizieren, die hinzugefügt oder entfernt werden müssen, muss zunächst die Vererbungskette der betroffenen Klasse im alten und neuen Schema ermittelt werden. Mithilfe der `CIMUtils.listSuperclasses`-Methode werden alle Superklassen aus dem alten und neuen Modell ausgelesen und als Sets gespeichert. Anschließend wird die Schnittmenge der beiden Sets gebildet, um gemeinsame Superklassen zu identifizieren und zu entfernen, da diese keine

Änderungen an den geerbten Properties verursachen. Nur das Entfernen der Schnittmenge reicht jedoch nicht aus, da es möglich ist, dass eine gemeinsame Superklasse umbenannt wurde. Daher muss zusätzlich einmal über die bestätigten Klassenumbenennungen iteriert werden, um auch diese aus den Sets zu entfernen. Code 5.13 zeigt, wie die Superklassen in der Methode `addPropertyChangesFromInheritance`-Methode ermittelt werden.

```
var oldUri = classChange.getOldIRI() != null ? classChange.getOldIRI() :
     classChange.getIri();
var oldSuperClasses = CIMClassUtils.listSuperClasses(
    oldModel.getResource(oldUri));
var newSuperClasses = CIMClassUtils.listSuperClasses(
    newModel.getResource(classChange.getIri()));

var commonSuperClasses = new HashSet<>(oldSuperClasses);
commonSuperClasses.retainAll(newSuperClasses);
oldSuperClasses.removeAll(commonSuperClasses);
newSuperClasses.removeAll(commonSuperClasses);
removeRenamedClassesFromSuperClassChanges(oldSuperClasses,
    newSuperClasses, classRenames, oldModel, newModel);
```

Code 5.13: Ermitteln der zu entfernenden und hinzuzufügenden Superklassen

Nachdem die Superklassen korrekt gefiltert wurden, können nun anhand der entfernten und hinzugefügten Superklassen die zu entfernenden und hinzuzufügenden Properties ermittelt werden. Die zu entfernenden Properties lassen sich einfach durch das Auflisten aller Properties der entfernten Superklassen im alten Modell ermitteln. Bei den hinzuzufügenden Properties muss zusätzlich geprüft werden, welche der Properties bereits vor dem Schemaupdate definiert waren und welche neu hinzugekommen sind, da alle neu hinzugefügten Properties bereits Change-Objekte besitzen. Würden diese Properties daher in diesem Schritt nicht ausgefiltert, würden sie auf jeder Instanz doppelt angelegt werden.

Um die neuen Change-Objekte auf den richtigen Klassen zu generieren, werden daraufhin ähnlich zu den Superklassen die erbenden Klassen für das alte und neue Modell ermittelt. Falls die Klasse, deren Superklasse geändert wurde, selbst instanziierbar ist, wird sie ebenfalls zu der Liste der erbenden Klassen hinzugefügt. An dieser Stelle wird die Liste der erbenden Klassen ähnlich wie bei den Superklassen gefiltert, diesmal jedoch anhand der neu hinzugefügten Klassen. Eine neu hinzugefügte Klasse kann schließlich noch keine Instanzen besitzen, die migriert werden müssen, weshalb diese Klasse bei Änderungen durch Vererbung ignoriert

werden kann.

Als letzter Schritt werden nun die beiden Hilfsmethoden `removeOldInherited-Properties` und `addNewInheritedProperties` aufgerufen, die jeweils die alten bzw. neuen Superklassen und erbenden Klassen sowie die zu entfernenden bzw. hinzuzufügenden Properties übergeben bekommen. Die beiden Methoden arbeiten beide ähnlich und unterscheiden sich nur in der Art der Change-Objekte, die sie erzeugen. Beide Methoden iterieren über die erbenden Klassen und prüfen, ob die Klasse instanziierbar ist. Falls nicht, kann die Klasse übersprungen werden, da keine Instanzen migriert werden müssen. Falls die Klasse instanziierbar ist, wird geprüft, ob die Klasse bereits ein Change-Objekt besitzt, welches modifiziert werden kann. Dafür wird die zuvor erstellte Map der Klassenänderungen genutzt, um nicht jedes Mal über alle Klassenänderungen iterieren zu müssen. Falls kein Change-Objekt existiert, wird ein neues `SemanticClassChange`-Objekt erstellt und sowohl der Liste der Klassenänderungen als auch der Map hinzugefügt.

Anschließend wird über alle zu entfernenden bzw. hinzuzufügenden Properties iteriert und für jede Property ein entsprechendes Change-Objekt erstellt. Zu löschende Properties werden mit dem `SemanticResourceChangeType REMOVED_-FROM_INHERITANCE` und hinzuzufügende Properties mit dem Typ `ADDED_FROM_-INHERITANCE` erstellt. Diese Change-Objekte werden dann der Liste der Property-Änderungen der jeweiligen Klassenänderung hinzugefügt.

5.5.2 Informationen zu Default-Werten

Damit der Nutzer im Frontend die passenden Default-Werte eingeben kann, müssen den Change-Objekten der Properties einige Informationen hinzugefügt werden. Dies geschieht in der `DefaultValueAssigner` Utility-Klasse, die ebenfalls vom `SchemaMigrationService` aufgerufen wird. Wie bei den Änderungen durch Vererbung können die benötigten Informationen nicht immer direkt aus den Change-Objekten abgeleitet werden, weshalb mit einem `Model` des neuen Schemas gearbeitet wird.

Die Hauptmethode der Klasse ist `assignDefaultValues`, die über alle Klassenänderungen iteriert und für jede Klasse eigene Methoden für die verschiedenen Arten von Property-Änderungen aufruft.

Die Methode `assignDefaultsToAttributes` filtert zunächst alle gelöschten Attribute heraus, da diese keine Default-Werte benötigen. Anschließend wird für jedes hinzugefügte oder geänderte Attribut die Methode `assignDefaultValueToAttribute` aufgerufen, die die Felder `optional`, `dataType` und `primitiveDatatype` des Change-Objekts füllt. Handelt es sich bei dem Datentyp um ein Enum, wird dabei die

Liste `allowedValues` auf die möglichen Werte des Enums gesetzt. Zudem wird geprüft, ob das Attribut einen Fixed- oder Default-Wert besitzt, der in diesem Fall genutzt wird, um das Feld `defaultValue` vorzubelegen. Code 5.14 zeigt wie die Felder `primitiveDatatype` und `dataType` anhand der verschiedenen Arten von Attributen befüllt werden.

```
if (CIMAttributeUtils.hasPrimitiveDatatype(attributeResource) ||
    CIMAttributeUtils.hasCIMDatatype(attributeResource)) {
    attributeChange.setPrimitiveDataType(
        CIMAttributeUtils.getPrimitiveDatatype(attributeResource).getURI
        ());
    attributeChange.setDataType(attributeResource.getProperty(
        CIMS.datatype).getResource().getURI());
} else if (CIMAttributeUtils.hasEnumAttribute(attributeResource)) {
    assignDefaultValueToEnumAttribute(attributeChange, attributeResource)
        ;
} else {
    var datatype = attributeResource.getProperty(CIMS.datatype)
        .getResource().getURI();
    attributeChange.setDataType(datatype);
    attributeChange.setPrimitiveDataType(datatype);
}
```

Code 5.14: Zuweisen der Datentyp-Informationen zu Attributen

`assignDefaultsToAssociations` filtert ebenfalls zunächst die gelöschten Assoziationen heraus und ruft für jede andere Assoziation die Methode `assignDefaultValueToAssociation` auf. Diese füllt auf gleichem Wege wie bei den Attributen die Felder `associationUsed` und `range` der `SemanticAssociationChange`-Objekte. Die Methode `assignDefaultsToEnumEntries` füllt schließlich für alle gelöschten Enum Entries die Liste `allowedValues` der `SemanticEnumEntryChange`-Objekte mit den verbleibenden Enum Entries des Enums.

Nachdem alle Änderungen durch Vererbung ermittelt und die benötigten Informationen zu den Default-Werten hinzugefügt wurden, können die Change-Objekte an das Frontend übergeben werden, wo der Nutzer die Default-Werte eingeben kann.

5.6 Generierung des SPARQL Skriptes

Nachdem der Benutzer alle Umbenennungen validiert und die benötigten Default-Werte angegeben hat, kann aus diesen Angaben das finale Migrationsskript generiert werden. Das Skript wird in Form mehrerer SPARQL-Update Anweisungen erstellt, die der Benutzer nach dem Herunterladen direkt auf seinen Instanzdaten ausführen kann.

Die Logik für die Skriptgenerierung ist dabei über mehrere Klassen verteilt. Wie alle anderen Anfragen aus dem Frontend, wird auch die Generierung des Migrationsskriptes von dem zentralen `SchemaMigrationService` angenommen und wie bei einigen vorherigen Schritten ist die eigentliche Logik für die Skriptgenerierung nicht in dem Service selbst enthalten, sondern in einer eigenen Utility-Klasse gekapselt. Allerdings ruft der Service in diesem Fall nicht direkt eine Methode dieser Klasse auf, sondern nutzt dafür das `MigrationBuilder` Interface. Die Nutzung dieses Interfaces entspricht dem Prinzip der hexagonalen Architektur und soll es ermöglichen, in Zukunft weitere Implementierungen für die Skriptgenerierung hinzuzufügen, die nicht auf SPARQL basieren.

Die Generierung des Skriptes ist in zwei Stufen unterteilt, die von zwei verschiedenen Klassen übernommen werden. Der `SPARQLMigrationBuilder`, der das `MigrationBuilder` Interface implementiert, orchestriert die Generierung des Skriptes und entscheidet, welche der gespeicherten Änderungen welche Updates zur Folge haben. Für die eigentliche Erstellung der SPARQL-Updates ruft er die `SPARQLUpdateGenerator`-Klasse auf, die aufgrund von parametrisierten Templates die einzelnen SPARQL-Updates erstellt.

Diese Aufteilung in zwei Stufen hält die beiden Klassen übersichtlich, indem sie die Logik der Skriptgenerierung von dem simplem Laden und Befüllen der Templates trennt.

5.6.1 SPARQLMigrationBuilder

Die einzige öffentliche Methode des `SPARQLMigrationBuilder` ist die `generateMigrationScript`-Methode, welche das `MigrationBuilder` Interfaces implementiert. Diese Methode bekommt die Liste der validierten `SemanticClassChange`-Objekte als Parameter übergeben. Nachdem diese Methode von dem `SchemaMigrationService` aufgerufen wurde, werden ähnlich zu Abschnitt 5.3 verschiedene private Methoden aufgerufen, die jeweils für die Verarbeitung einer Art von Ressource zuständig sind.

Code 5.15 zeigt beispielhaft die Methode für die Verarbeitung von Klassenänderun-

gen. Zuerst wird mittels eines Switch-Statements der `SemanticResourceChangeType` der Ressource ermittelt und gegebenenfalls ein Update generiert. Ein Update muss generiert werden, falls die Klasse entweder gelöscht oder umbenannt wurde. Mögliche Änderungstypen, die keine direkten Updates erfordern, wie z.B. `CHANGE`, werden in dem Switch-Statement nicht aufgelistet.

```
private String generateUpdateForClass(SemanticClassChange classChange) {
    var result = new UpdateRequest();

    String update = switch (classChange.getSemanticResourceChangeType())
        {
        case DELETE -> updateGenerator.generateDeleteClassUpdate(
            classChange);
        case RENAME -> updateGenerator.generateRenameClassUpdate(
            classChange);
        default -> null;
    };

    if (update != null) {
        result.add(update);
    }

    if (classChange.getSemanticResourceChangeType() !=
        SemanticResourceChangeType.DELETE &&
        classChange.getSemanticResourceChangeType() !=
            SemanticResourceChangeType.ADD) {
        processClassChanges(classChange, result);
    }

    processPropertyChanges(classChange, result);

    return result.toString();
}
```

Code 5.15: Methode für Erstellung von Klassen-Updates

Anschließend müssen noch die Änderungen an den Eigenschaften der Ressource betrachtet werden. Hierfür wird jeweils eine eigene Methode aufgerufen, die wieder mittels eines Switch-Statements den `SemanticFieldChangeType` der einzelnen Änderung ermittelt und die generierten Updates aggregiert. Dieses Verarbeiten kann in gewissen Fällen allerdings auch übersprungen werden, um unnötige Updates zu vermeiden. Das Update, welches bei einer Löschung einer Ressource generiert

wird, entfernt beispielsweise alle Tripel, die diese Ressource als Subjekt haben. Ein separates Löschen der Eigenschaften der Ressource ist in diesem Fall also nicht notwendig. Daher wird in Code 5.15 das Verarbeiten der Klassen-Änderungen nur durchgeführt, wenn die Klasse weder gelöscht noch hinzugefügt wurde.

Im spezifischen Fall von Klassenänderungen müssen nach dem Verarbeiten der Änderungen der Eigenschaften zudem noch die Properties verarbeitet werden. Abschließend wird das aggregierte Update an die aufrufende Methode zurückgeben. Hierbei muss das Update zuerst in einen String umgewandelt werden, da sich nur Strings zu einem UpdateRequest-Objekt hinzufügen lassen.

5.6.2 SPARQLUpdateGenerator

Der `SPARQLUpdateGenerator` enthält einzelne Methoden für jede Art von SPARQL-Update, die im Migrationsskript enthalten sein kann. Die Methoden nutzen jeweils den `SPARQLTemplateLoader`, um das Template für das entsprechende Update aus einer Datei auszulesen und als einen `ParameterizedSPARQLString` zu speichern. Code 5.16 zeigt beispielhaft das Template für das Löschen eines Enum Entry. Das Update soll in diesem Fall alle Tripel finden, in denen ein Attribut das gelöschte Enum Entry als Wert hat, und diese Vorkommen durch einen neuen Wert ersetzen. Um das Template zu befüllen, müssen also zwei Parameter gesetzt werden: `?enumEntry` für den zu löschenden Enum Wert und `?replacementValue` für den neuen Wert.

```
DELETE {
    ?instance ?attribute ?enumEntry .
} INSERT {
    ?instance ?attribute ?replacementValue .
} WHERE {
    ?instance ?attribute ?enumEntry .
}
```

Code 5.16: Template für das Löschen eines Enum Entrys

Das Ersetzen der Parameter kann mit Apache Jenas `ParameterizedSPARQL-Strings` durch einen `setIRI` oder `setLiteral` Aufruf gemacht werden. Code 5.17 zeigt wie die Parameter des in Code 5.16 gezeigten Templates mithilfe von `setIRI` Aufrufen ersetzt werden. Die Nutzung von `ParameterizedSPARQLStrings` gegenüber dem einfachen String-Ersetzen hat hierbei den Vorteil, dass Jena die korrekte Formatierung der IRIs und Literale sicherstellt, was besonders bei getypten Literalen hilfreich ist.

```
public String generateDeleteEnumEntryUpdate(SemanticEnumEntryChange
    enumEntryChange) {
    var pss = SparqlTemplateLoader.loadTemplate("migration/enum-entry-
        deleted");
    pss.setIri("enumEntry", enumEntryChange.getIri());
    pss.setIri("replacementValue", enumEntryChange.getReplacementValue())
        ;
    return pss.toString();
}
```

Code 5.17: Beispiel für das Generieren von SPARQL-Updates

5.7 Frontend Implementierung

Die Implementierung des Web-Wizards im Frontend ist in mehrere Komponenten unterteilt. Anhand des File-based Routings von SvelteKit wurde zunächst ein neuer Ordner `migrate` mit einer `+page.svelte` Datei erstellt, wodurch die Route `/migrate` für den Migrations-Wizard zur Verfügung steht. Diese Hauptkomponente ist für die Darstellung des in Abschnitt 4.3 beschriebenen Layouts und der Navigation zwischen den Schritten verantwortlich. Zusätzlich wird für jeden Schritt und potentiellen Unterschritt des Wizards eine eigene Svelte-Komponente erstellt, die jeweils die spezifische Logik und Darstellung des Schrittes übernimmt.

5.7.1 Navigation

Um das Einbinden der verschiedenen Komponenten und die Navigation zwischen den Schritten zu behandeln, wurde in der `+page.svelte` Datei ein `steps`-Array angelegt, welches die Information für die einzelnen Schritte des Wizards enthält. Jeder Schritt ist dabei als Objekt mit den Eigenschaften `title` und `component` definiert. Der `title` enthält eine kurze Beschreibung des Schrittes, die in der Übersichtsleiste angezeigt werden kann, während `component` die Komponente referenziert, die für die Darstellung des jeweiligen Schrittes verantwortlich ist. Genauer gesagt enthält `component` jeweils eine Variable, die auf den Konstruktor der jeweiligen Komponente zeigt. Falls ein Schritt noch weitere Unterschritte besitzt, wie bei den Property-Umbenennungen, werden diese in einem zusätzlichen `substeps`-Array innerhalb des Schrittes definiert.

Zusätzlich werden für die Navigation einige Hilfsvariablen definiert. `currentStepIndex` und `currentSubstepIndex` speichern die aktuellen Indizes des `steps`- und `substeps`-Arrays, um die Navigation zwischen den Schritten zu ermöglichen. Die

Variable `currentSteps` wird mithilfe einer derived-Funktion als Eintrag aus dem `steps`-Array basierend auf dem aktuellen Index berechnet und `stepInstance` speichert die Instanz des aktuell angezeigten Schrittes.

Mithilfe dieser Variablen kann nun die jeweilige Komponente des aktuellen Schrittes dynamisch in der Hauptkomponente eingebunden werden (s. Code 5.18)

```
<div class="no-scrollbar h-full overflow-y-scroll">
    <currentStep.component
        bind: this={stepInstance}
        substeps={currentStep.substeps}
        currentSubstepIndex={currentSubstepIndex}
        bind: disableNext
    />
</div>
```

Code 5.18: Dynamisches Einbinden der aktuellen Schritt-Komponente

Svelte erlaubt die direkte Nutzung von Komponenten-Konstruktoren in Markup, wodurch `currentStep.component` dynamisch die jeweilige Komponente des aktuellen Schrittes rendert. Zusätzlich werden einige Properties an die Schritt-Komponente übergeben, darunter ein potentielles `substeps`-Array und der `currentSubstepIndex`, um die Unterschritte zu verwalten. Außerdem wird die Variable `disableNext` gebunden, die von der Schritt-Komponente gesetzt werden kann, um die „Continue“-Schaltfläche zu deaktivieren, falls der Nutzer noch nicht alle erforderlichen Eingaben gemacht hat. Sowohl Unterschritte als auch das Blockieren der „Continue“-Schaltfläche werden nicht in allen Schritten benötigt, allerdings führt das Übergeben von nicht deklarierten Properties in Svelte nicht zu einem Fehler, weshalb dies hier unproblematisch ist. Zuletzt wird die Instanz der aktuellen Schritt-Komponente an die `stepInstance`-Variable gebunden, um aus der Hauptkomponente Methoden der Schritt-Komponente aufrufen zu können.

Um nun zwischen den verschiedenen Schritten navigieren zu können, sind für die „Back“- und „Continue“-Buttons in der Hauptkomponente die Methoden `handleBackButton` und `handleContinueButton` registriert. Beide Methoden prüfen jeweils, ob der aktuelle Schritt Unterschritte besitzt und passen die beiden Variablen `currentStepIndex` und `currentSubstepIndex` entsprechend an, um entweder zum vorherigen oder nächsten Schritt oder Unterschritt zu navigieren. Außerdem wird jeweils `disableNext` zurückgesetzt, um die „Continue“-Schaltfläche im neuen Schritt wieder zu aktivieren.

5.7.2 Kommunikation mit dem Backend

Das Anfragen und Absenden von Daten an das Backend erfolgt in jedem der Schritte gleich. Hierfür wird auf jeder der Schritt-Komponenten eine Methode `onNext` und ein `onMount` definiert. Die `onMount`-Methode wird automatisch von Svelte aufgerufen, wenn die Komponente von der Hauptkomponente eingebunden wird, und enthält in allen Schritten außer dem ersten eine GET-Anfrage an das Backend, um die für den Schritt benötigten Daten zu laden.

Die `onNext`-Methode hingegen wird aus der `handleContinueButton`-Methode der Hauptkomponente aufgerufen und enthält in allen Schritten außer dem letzten eine POST-Anfrage, die die Nutzereingaben des Schrittes zurück an das Backend sendet. Der letzte Schritt, der das Migrationsskript generiert, bildet eine Ausnahme, da hier keine weiteren Nutzereingaben mehr gemacht werden können. Stattdessen enthält die Seite einen Button, der beim Klicken eine GET-Anfrage an das Backend sendet, um das generierte Migrationsskript herunterzuladen. Die `onNext`-Methode dieses Schrittes enthält nur noch eine Weiterleitung zurück zur Startseite nach dem Herunterladen des Skriptes.

5.7.3 MigrationState

Die Schemamigration ist in ihrer Nutzung sehr eng mit der bereits implementierten Vergleichsansicht verbunden und es ist wahrscheinlich, dass sich Nutzer vor der Erstellung eines Migrationsskriptes die Unterschiede der beiden Schemaversionen ansehen möchten. Daher wurde auf der Schemavergleichsansicht ein Button hinzugefügt, der den Nutzer direkt zum Migrations-Wizard weiterleitet. Allerdings würden bei einer einfachen Weiterleitung die für den Vergleich ausgewählten Schemata verloren gehen und der Nutzer müsste sie erneut auswählen.

Um dies zu verhindern, wurde ein globaler Store namens `MigrationStore` erstellt, der genutzt werden kann, um die ausgewählten Schemata zwischenzuspeichern und die Schemaauswahl des Mirationswizards vorzubelegen. Der Store speichert dabei den Datensatz und die IRI des neuen Schemas und eine hochgeladene Datei des alten Schemas. Diese Werte werden in der `onMount`-Methode des ersten Schrittes aus dem Store geladen und die Auswahlfelder entsprechend vorbelegt. Dadurch muss der Nutzer die Schemata nicht erneut auswählen, wenn er vom Schemavergleich zum Migrations-Wizard wechselt, hat aber weiterhin die Möglichkeit Änderungen vorzunehmen, falls er andere Schemata migrieren möchte. In der `onNext` Methode des letzten Schrittes wird der Store anschließend zurückgesetzt, um die Daten zu löschen.

6 Fazit und Ausblick

Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und evaluiert, inwiefern die anfangs formulierten Ziele erreicht wurden. Zudem werden die Limitationen der entwickelten Implementierung diskutiert und ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben.

6.1 Bewertung der Zielerreichung

Im Rahmen dieser Arbeit konnte erfolgreich eine Erweiterung des RDF-Architects umgesetzt werden, welche die automatisierte Generierung von SPARQL-Skripten zur Migration von Instanzdaten nach einem Schemaupdate ermöglicht. Die Implementierung erspart dadurch das manuelle Erstellen von Migrationsskripten und macht den Prozess effizienter und weniger fehleranfällig.

Anhand der Definition der CIM-Ressourcen konnten die verschiedenen Änderungsfälle identifiziert und die daraus hervorgehenden Migrationen abgeleitet werden. Bestehende Konzepte zur Umbenennungserkennung in Ontologien wurden betrachtet und es wurde ein eigener Ansatz entwickelt, der auf den spezifischen Anforderungen des CIM-Kontexts basiert. Dieser Ansatz zeigt anhand von synthetischen Testfällen und dem Testen mittels echten Schemata in der UI eine gute Erkennungsrate für Umbenennungen. Der Einsatz in produktiven Projekten ist geplant.

Die Implementierung im Backend des RDF-Architects umfasst die Analyse der semantischen Änderungen, die Erkennung von Umbenennungen der Ressourcen sowie die Generierung der entsprechenden SPARQL-Updates. Die Analyse der Änderungen ist dabei in der Lage, beinahe alle der erfassten Änderungsfälle zu erkennen. Einzig das Verschieben von Properties zwischen Klassen kann aufgrund der Entscheidung, Property-Vergleiche auf ihre Domain zu begrenzen, nicht erkannt werden. Hingegen kann das Ändern der Multiplizität von Assoziationen zwar erkannt, allerdings aufgrund der Trennung von Schema- und Instanzdaten im RDF-Architect nicht vollständig korrekt migriert werden. Insgesamt ist die Implementierung jedoch in der Lage, einen Großteil der Änderungen zu erkennen und zuverlässige Migrationsskripte zu erstellen. Gegebenenfalls sind bei den Sonderfällen und komplexen Migration manuelle Ergänzungen der Skripte notwendig

Für das Frontend konnte durch die Nutzung des Wizard-Patterns ein benutzerfreundliches Interface geschaffen werden, welches eine gute Balance zwischen Übersichtlichkeit und Funktionalität bietet. Das Interface bietet mehrere Möglichkeiten

zur Anpassung der Migration, wie das Überprüfen von Umbenennungen oder das Festlegen von Default-Werten, bleibt dabei jedoch intuitiv und einfach zu bedienen.

6.2 Limitationen der Implementierung

Die meisten Limitationen dieser Implementierung ergeben sich durch die Entscheidung, den RDF-Architect als Modellierungssoftware von den Instanzdaten der Schemata zu trennen. Die größte Schwachstelle besteht dabei in der Behandlung von Assoziationen. Da zur Zeit der Erstellung des Migrationsskriptes nicht bekannt ist, welche Instanzen vorliegen, können neue oder geänderte Assoziationen nur mittels SPARQL-Patterns zugewiesen werden. Fehler bei dieser Zuweisung oder bei fehlgeschlagenen Typkonvertierungen können dadurch auch nicht von der Anwendung selbst festgestellt werden, sondern müssen vom Nutzer selbst mittels SHACL Validierung erkannt und behoben werden.

Eine weitere Einschränkung entsteht durch die Entscheidung, die Umbenennungserkennung von Properties auf die eigene Domain zu begrenzen. Der Vergleich muss dadurch zwar deutlich weniger Properties in Betracht ziehen und ist daher performanter, kann allerdings auch keine Verschiebungen von Properties erkennen. Wird beispielsweise ein Property von einer Klasse auf seine Superklasse verschoben, wird dies als Löschen und Hinzufügen erkannt, wodurch Werte auf bestehenden Instanzen nicht beibehalten werden können.

6.3 Ausblick

Es bestehenden mehrere Möglichkeiten zur Weiterentwicklung der in dieser Arbeit entwickelten Schemamigration. Eine Option wäre, die Schemamigration vom RDF-Architect zu entkoppeln und entweder als externe Bibliothek einzubinden oder als separate Software zu Verfügung zu stellen. Dadurch ließe sich die tatsächliche Durchführung der Migration in den Prozess integrieren und es könnten auf Basis der Instanzdaten fundiertere Entscheidungen getroffen werden.

Zudem könnten weitere Funktionalitäten ergänzt werden, wie etwa die Erkennung von Verschiebungen von Properties zwischen Klassen oder die Behandlung von komplexeren Änderungen wie das Zusammenführen oder Aufteilen von Klassen. Im Fall eines Aufteilens einer Klasse müssten dabei erneut SPARQL-Patterns genutzt werden, um zu entscheiden, welcher neuen Klasse eine Instanz zugewiesen werden soll. Auch die Zuweisung von Default-Werten für Attribute könnte mithilfe von SPARQL-Patterns verbessert werden, um Werte aus anderen Eigenschaften der

Instanz abzuleiten.

Ein weiterer Punkt der verbessert werden könnte, ist die Formel, die zur Berechnung der Ähnlichkeit der Ressourcen genutzt wird. Insbesondere die Gewichtung der strukturellen Eigenschaften, welche für diese Arbeit alle gleich gewichtet wurden, könnte anhand ausgiebiger Tests weiter analysiert und optimiert werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literatur

[1] „A Brief History: The Common Information Model.“ Archivierte Version verfügbar unter https://web.archive.org/web/20220602142235/https://site.ieee.org/pes-enews/2015/12/10/a-brief-history-the-common-information-model/, besucht am 12. Dez. 2025. Adresse: https://site.ieee.org/pes-enews/2015/12/10/a-brief-history-the-common-information-model/

[2] „Chapter 3: CIM Background,“ besucht am 10. Feb. 2026. Adresse: https://author-msites.epri.com/rd/research/062333/common-information-model-primer/chapter-3-cim-background#4257225834-2628575798

[3] M. Gietz und T. Rogowski, „CGMES as an interface for multilateral grid modelling data exchange,“ in *2019 Modern Electric Power Systems (MEPS)*, 2019.

[4] „RDF 1.1 Concepts and Abstract Syntax,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts

[5] „Resources and Statements,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#resources-and-statements

[6] „Triples,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-triples

[7] „IRIs,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-IRIs

[8] „vocabularies and Namespace IRIs,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#vocabularies

[9] „Literals,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal

[10] „Blank Nodes,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf11-concepts/#section-blank-nodes

[11] „RDF Schema 1.1,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/rdf-schema/

[12] „RDF 1.1 Turtle,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/turtle/

[13] „Making Simple Queries (Informative),“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-query/#basicpatterns

[14] „Graph Patterns,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-query/#GraphPattern

[15] „SELECT,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-query/#select

[16] „Graph Update,“ besucht am 10. Feb. 2026. Adresse: https://www.w3.org/TR/sparql11-update/#graphUpdate

[17] „ENTSO-E Mission Statement,“ besucht am 10. Feb. 2026. Adresse: https://www.entsoe.eu/about/inside-entsoe/mission-statement/

[18] „Common Grid Model Exchange Specification (CGMES) Version 2.5,“ besucht am 10. Feb. 2026. Adresse: https://eepublicdownloads.entsoe.eu/clean-documents/CIM_documents/IOP/CGMES_2_5_TechnicalSpecification_61970-600_Part%201_Ed2.pdf

[19] „What is SHACL?“ Besucht am 10. Feb. 2026. Adresse: https://www.ontotext.com/knowledgehub/fundamentals/what-is-shacl/

[20] „Validation Report,“ besucht am 5. Feb. 2026. Adresse: https://www.w3.org/TR/shacl/#validation-report

[21] W. W. Cohen, P. Ravikumar, S. E. Fienberg u. a., „A comparison of string distance metrics for name-matching tasks.,“ in *IIWeb*, Bd. 3, 2003, S. 73–78.

[22] R. W. Hamming, „Error detecting and error correcting codes,“ *The Bell system technical journal*, Jg. 29, Nr. 2, S. 147–160, 1950.

[23] V. Levenshtein, „Binary codes capable of correcting deletions, insertions, and reversals,“ in *Soviet physics-doklady*, Bd. 10, 1966.

[24] M. A. Jaro, „Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida,“ *Journal of the American Statistical association*, Jg. 84, Nr. 406, S. 414–420, 1989.

[25] W. E. Winkler, „String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage.,“ 1990.

[26] P. Jaccard, „Etude comparative de la distribution florale dans une portion des Alpes et des Jura,“ *Bull Soc Vaudoise Sci Nat*, Jg. 37, S. 547–579, 1901.

[27] G. Salton, A. Wong und C.-S. Yang, „A vector space model for automatic indexing,“ *Communications of the ACM*, Jg. 18, Nr. 11, S. 613–620, 1975.

[28] „Spring Framework Overview,“ besucht am 10. Feb. 2026. Adresse: https://docs.spring.io/spring-framework/reference/overview.html

[29] „Spring Boot Overview,“ besucht am 10. Feb. 2026. Adresse: https://docs.spring.io/spring-boot/index.html

[30] „Spring Boot Request Mapping,“ besucht am 10. Feb. 2026. Adresse: https://docs.spring.io/spring-framework/reference/web/webmvc/mvc-controller/ann-requestmapping.html

[31] „Apache Jena Documentation Overview,“ besucht am 10. Feb. 2026. Adresse: https://jena.apache.org/documentation/index.html

[32] „Apache Commons Documentation,“ besucht am 10. Feb. 2026. Adresse: `https://commons.apache.org/index.html`

[33] „Project Lombok,“ besucht am 10. Feb. 2026. Adresse: `https://projectlombok.org/`

[34] „Project Lombok Features,“ besucht am 10. Feb. 2026. Adresse: `https://projectlombok.org/features/`

[35] R. Harris. „Virtual DOM is pure overhead,“ besucht am 10. Feb. 2026. Adresse: `https://svelte.dev/blog/virtual-dom-is-pure-overhead`

[36] „What are Runes?“ Besucht am 10. Feb. 2026. Adresse: `https://svelte.dev/docs/svelte/what-are-runes`

[37] „Svelte Kit Introduction,“ besucht am 10. Feb. 2026. Adresse: `https://svelte.dev/docs/kit/introduction`

[38] „Tailwind CSS Overview,“ besucht am 10. Feb. 2026. Adresse: `https://tailwindcss.com/`

[39] „Tailwind Utility Classes,“ besucht am 10. Feb. 2026. Adresse: `https://tailwindcss.com/docs/styling-with-utility-classes`

[40] A. Cockburn, „Hexagonal architecture,“ *The Pattern: Ports and Adapters*, 2005.

[41] F. Ardjani, D. Bouchiha und M. Malki, „Ontology-alignment techniques: survey and analysis,“ *International Journal of Modern Education and Computer Science*, Jg. 7, Nr. 11, S. 67, 2015.

A Anhang

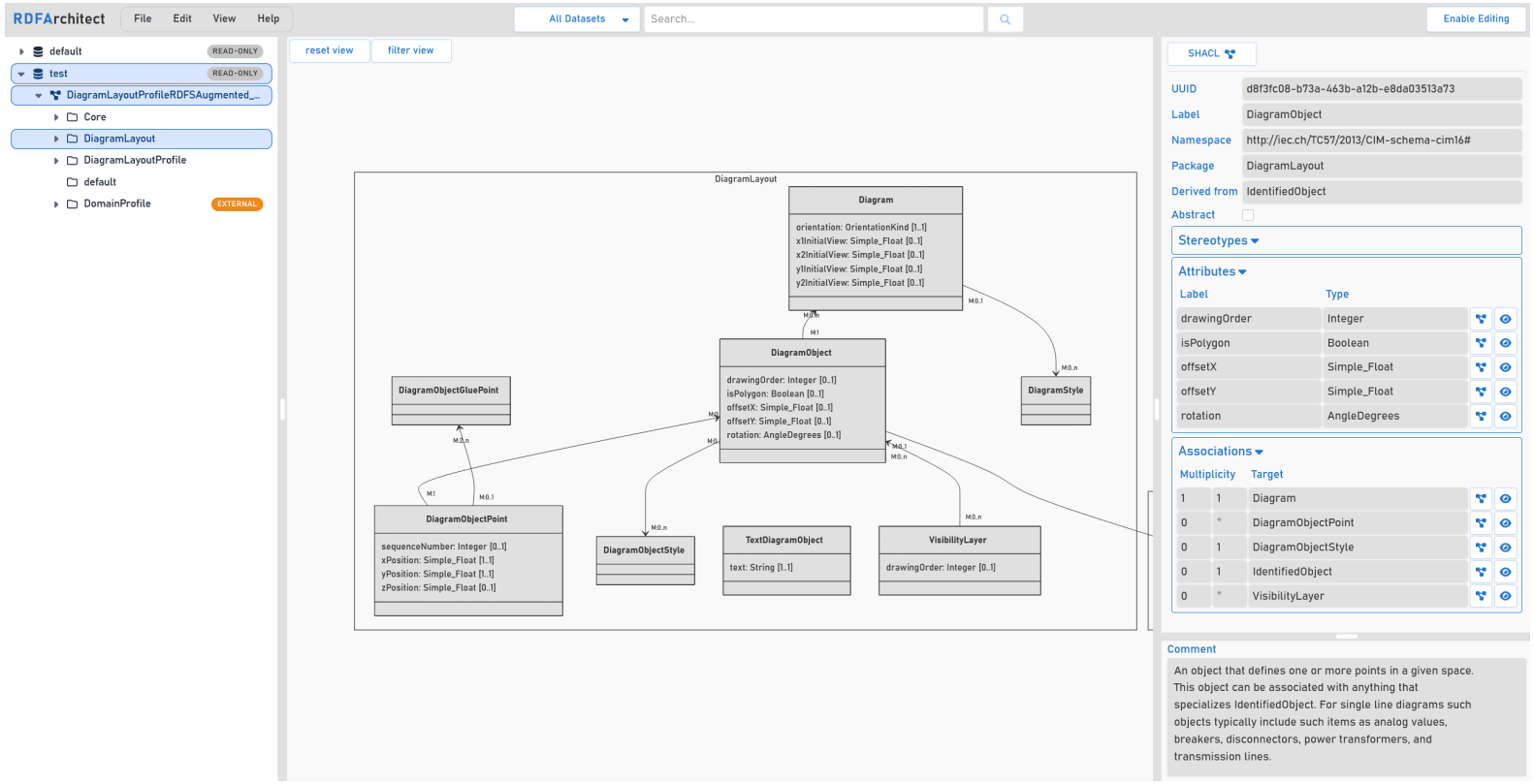


Abbildung A.1: RDF-Architect Übersicht (vergrößert dargestellt)

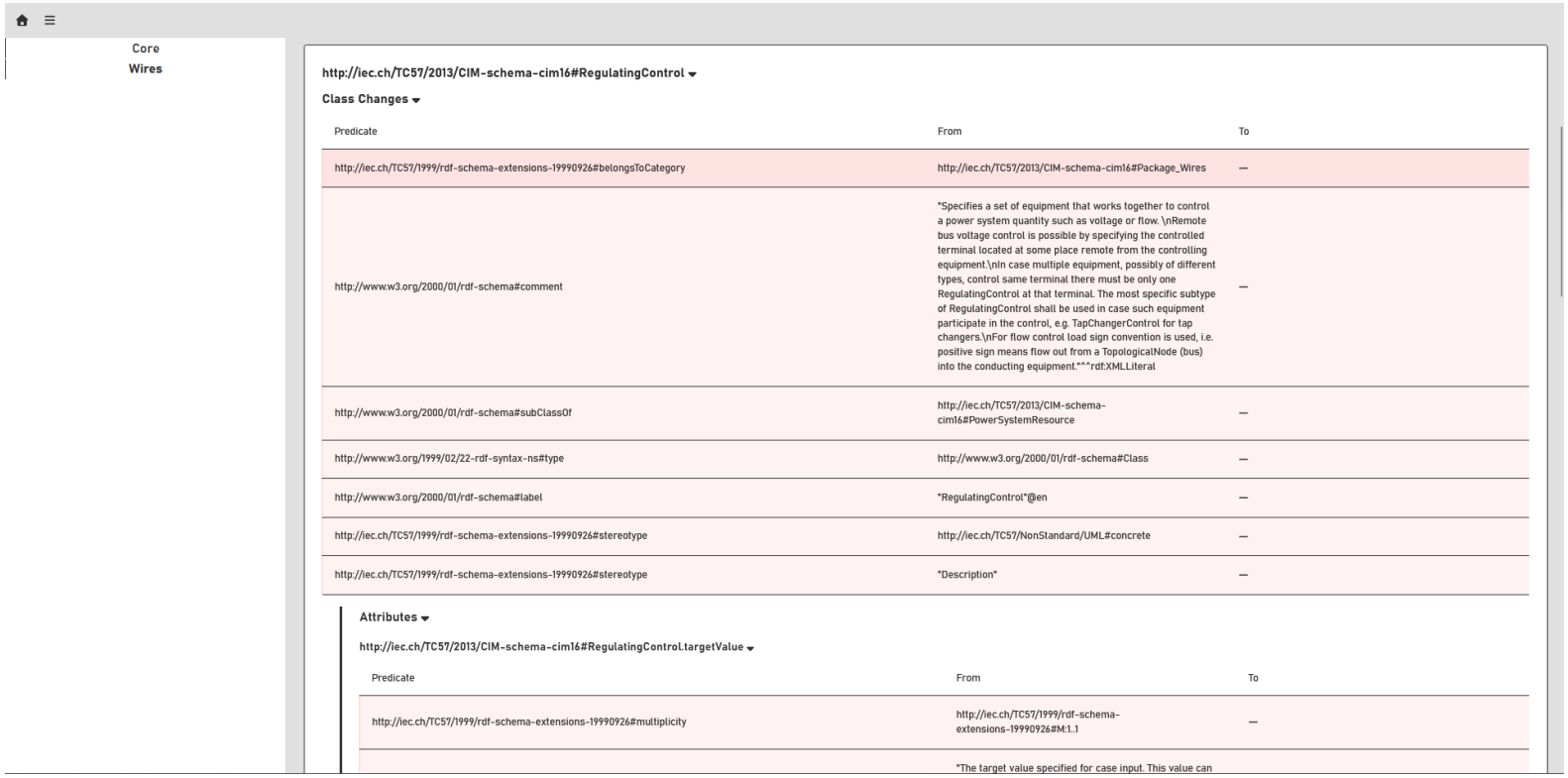


Abbildung A.2: Schemavergleich (vergrößert dargestellt)

Tabelle A.1: Label Testfälle

Ursprungslabel	**Umbenennungskandidaten**	**tatsächliche Umbenennung**	**Änderungstyp**
ActivePower	ReactivePower, PowerActive, ActiveEnergy, PowerRate		keine Umbenennung
CapacitivePower	CapacitiveReactor, ReactiveCapacity, PowerCapacitor, CapacitorPower, CapacitiveLoad		keine Umbenennung
BreakerStatus	BreakerState, StatusBreaker, SwitchStatus, BreakerSwitch, BreakerRelay		keine Umbenennung
NominalVoltage	VoltageNominal, NominalVolume, TerminalVoltage, VoltageRating, NominalCurrent		keine Umbenennung
FrequencyValue	ValueFrequency, FrequencyRange, FrequencyLevel, FrequencyUpdate, SequencyValue		keine Umbenennung
LoadCapacity	LoadCapabilities, LoadCapacitor, Capacity, PowerCapacity, VoltageLevel, Current, Transformer		keine Umbenennung
PhaseAngle	AngleDegree, PhaseShift, AngleMeasurement, VoltageLevel, Current, Transformer		keine Umbenennung
TransformerRatio	PowerRatio, FlowRate, TransformationRate, TransformerSwitch, VoltageLevel, Current		keine Umbenennung
CurrentPhase	PhaseType, PhaseCode, Voltage, VoltageLevel, Current, Transformer		keine Umbenennung
LoadProfile	Profile, Load, LoadPattern, PowerProfile, VoltageLevel, Current, Transformer		keine Umbenennung
PowerTransformerEnd	TerminalPower, LineVoltage, TransformerPower, VoltageLimit, Current, EquipmentStatus, PhaseVoltage, TransformerPowerEnd	TransformerPowerEnd	Wortreihenfolge geändert
ACLineSegment	PowerLine, LinePoint, LineSegment, ACTransformer, Current, Transformer, PowerFactor, LineSegmentAC	LineSegmentAC	Wortreihenfolge geändert
PowerFactor	ApparentPowerFactor, VoltageLevel, Current, LineVoltage, PhaseVoltage, EquipmentStatus, FactorPower	FactorPower	Wortreihenfolge geändert
VoltageCurrent	VoltageLevel, LineVoltage, CurrentFlow, Transformer, PhaseVoltage, PowerFactor, CurrentVoltage	CurrentVoltage	Wortreihenfolge geändert

Fortsetzung auf nächster Seite

Tabelle A.1 – *Fortsetzung*

Ursprungslabel	**Umbenennungskandidaten**	**tatsächliche Umbenennung**	**Änderungstyp**
LoadPower	LoadCapacity, PowerFactor, ActivePower, VoltageLevel, Current, Transformer, PowerLoad	PowerLoad	Wortreihenfolge geändert
VoltageLmit	CurrentLimit, TemperatureValue, VoltageLevel, LineVoltage, Transformer, ActivePower, VoltageLimit	VoltageLimit	Tippfehler
EquipmentStatu	DeviceStatus, VoltageLevel, PhaseVoltage, Equipment, LineVoltage, PowerFactor, EquipmentStatus	EquipmentStatus	Tippfehler
Transfomer	PowerTransformer, VoltageLevel, Current, PhaseVoltage, TemperatureValue, ActivePower, LineVoltage, Transformer	Transformer	Tippfehler
ActiveCurent	ActiveLine, VoltageLevel, CurrentFlow, Transformer, PowerFactor, LineVoltage, TemperatureValue, ActiveCurrent	ActiveCurrent	Tippfehler
Temprature	TemperatureLimit, TemperatureLimit, VoltageLevel, Current, Transformer, Temperature	Temperature	Tippfehler
VoltLvl	VoltageLimit, VoltageStep, Transformer, Velocity, PhaseVoltage, LineVoltage, ActivePower, VoltageLevel	VoltageLevel	Abkürzung
EquipStat	DeviceStatus, VoltageLevel, Prerequisite, Transformer, PhaseVoltage, LineVoltage, EquipmentStatus	EquipmentStatus	Abkürzung
TempLmt	TemperatureLevel, TempRange, VoltageLevel, Current, Transformer, PhaseVoltage, LineVoltage, TemperatureLimit	TemperatureLimit	Abkürzung
PwrTrans	VoltageLevel, Transfer, PowerDraw, Transmission, Transformer, PhaseVoltage, LineVoltage, PowerTransformer	PowerTransformer	Abkürzung
CurrLmt	CurrentLevel, CurrentValue, VoltageLevel, Transformer, PhaseVoltage, LineVoltage, PowerFactor, CurrentLimit	CurrentLimit	Abkürzung
Str	StringValue, Stream, VoltageLevel, Current, Transformer, PhaseVoltage, LineVoltage, String	String	Langform
Num	Numeric, NumericValue, VoltageLevel, Current, Transformer, PhaseVoltage, LineVoltage, Number	Number	Langform
Val	Validator, Validation, VoltageLevel, Current, Level, PhaseVoltage, LineVoltage, Value	Value	Langform
Freq	FrequencyLimit, FrequencyValue, VoltageLevel, Requirement, Transformer, PhaseVoltage, LineVoltage, Frequency	Frequency	Langform

Fortsetzung auf nächster Seite

Tabelle A.1 – *Fortsetzung*

Ursprungslabel	**Umbenennungskandidaten**	**tatsächliche Umbenennung**	**Änderungstyp**
Meas	Measurable, MeasurementValue, VoltageLevel, Seam, Transformer, PhaseVoltage, LineVoltage, Measurement	Measurement	Langform
mVoltageLevel	myVoltageLevel, MemberVoltageLevel, Transformer, Current, PhaseVoltage, LineVoltage, PowerFactor, VoltageLevel	VoltageLevel	Präfix entfernt
pCurrent	privateCurrent, PhaseVoltage, Transformer, VoltageLevel, LineVoltage, PowerFactor, ActivePower, Current	Current	Präfix entfernt
mainTransformer	PowerTransformer, VoltageLevel, Transformation, PhaseVoltage, LineVoltage, ActivePower, Transformer	Transformer	Präfix entfernt
oldVoltageLimit	newVoltageLimit, previousVoltageLimit, VoltageLevel, Current, Transformer, PhaseVoltage, VoltageLimit	VoltageLimit	Präfix entfernt
tmpPowerFactor	PowerFactorValue, tmpPowerDraw, VoltageLevel, Current, Transformer, curPowerFactor, LineVoltage, PowerFactor	PowerFactor	Präfix entfernt
VoltageValue	VoltageLevel, VoltageLimit, Transformer, Current, PhaseVoltage, LineVoltage, PowerFactor, Voltage	Voltage	Suffix entfernt
CurrentMeasurement	CurrentValue, LineCurrent, Transformer, VoltageLevel, PhaseVoltage, LineVoltage, PowerFactor, Current	Current	Suffix entfernt
PowerData	ActivePower, PowerFactor, Transformer, VoltageLevel, Current, PhaseVoltage, LineVoltage, Power	Power	Suffix entfernt
TransformerInfo	TransformerData, TransformerDetails, PowerTransformer, VoltageLevel, Current, PhaseVoltage, Transformer	Transformer	Suffix entfernt
FrequencyValue	FrequencyLimit, FrequencyRate, PowerFrequency, VoltageLevel, Current, Transformer, PhaseVoltage, Frequency	Frequency	Suffix entfernt
CircuitBreakerStatus	CircuitStatus, BreakerState, CircuitProtection, VoltageLevel, Current, Transformer, SwitchStatus	SwitchStatus	Fachbegriff geändert
ConductorWire	ConducingWire, ConductorSegment, WireConductor, VoltageLevel, Current, Transformer, CableWire	CableWire	Fachbegriff geändert
SubstationEquipment	SubstationDevice, EquipmentStation, StationEquipment, VoltageLevel, Current, Transformer, PowerPlantEquipment	PowerPlantEquipment	Fachbegriff geändert
GeneratingUnit	GeneratingDevice, Generator, VoltageLevel, Current, Transformer, PowerSourceUnit	PowerSourceUnit	Fachbegriff geändert

Fortsetzung auf nächster Seite

Tabelle A.1 – *Fortsetzung*

Ursprungslabel	**Umbenennungskandidaten**	**tatsächliche Umbenennung**	**Änderungstyp**
FeederLine	FeederSegment, LineSegment, VoltageLevel, Current, Transformer, DistributionLine	DistributionLine	Fachbegriff geändert