TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Bauingenieurwesen Institut für Bauinformatik

Projektarbeit

Computergestützte Visualisierung und Verwaltung von Model View Definitions für BIM im Bahnbau

Computer-based visualization and management of Model View Definitions for BIM in railway-projects

eingereicht von
Christoph Mellüh
geb. am 10.09.1998 in Weimar
Matrikel-Nummer: 4720835

Betreuer/in:

- Prof. Dr.-Ing. habil. Karsten Menzel
- Dipl.-Ing Al-Hakam Hamdan
- Dip.-Ing Frank Opitz

Dresden, den 23.03.2022

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Bauingenieurwesen Institut für Bauinformatik

SELBSTSTÄNDIGKEITSERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich reiche sie erstmals als Prüfungsleistung ein. Mir ist bekannt, dass ein Betrugsversuch mit der Note „nicht ausreichend“ (5,0) geahndet wird und im Wiederholungsfall zum Ausschluss von der Erbringung weiterer Prüfungsleistungen führen kann.

Name: Mellüh

Vorname: Christoph

Matrikelnummer: 4720835

Dresden, den 23.03.2022

Unterschrift Christoph Mellüh

I ABSTRAKT

Mit der vorliegenden Projektarbeit wird eine Möglichkeit geschaffen, Model View Definitions (MVD) zur Filtrierung und Prüfung von IFC-Modellen einem breiteren Publikum zugänglich zu machen. Obwohl das Konzept der MVD bereits seit 2012 von buildingSMART empfohlen wird (Chipman et al., 2012), um einen Datenaustausch mithilfe von IFCs zu unterstützen, ist das zugehörige Dateiformat, die mvdXML, bisher kaum im Bereich des Endanwendenden angelangt.

Laut Krijnen & Luttun (2021) ist ein bekanntes Problem von MVD und IFC, dass fachfremde Personen oft die erstellten Daten nicht unabhängig/selbstständig benutzen können, sondern auf externe Hilfe von Spezialisten angewiesen sind. Durch die komplexe Struktur von mvdXML-Dateien ist es für Personen ohne IT-Hintergrund oft nicht möglich die geforderten Konzepte zu verstehen und entsprechend umzusetzen bzw. anzupassen. Hier versucht die Projektarbeit anzugreifen und eine grafische Benutzeroberfläche zu schaffen, welche es fachfremden Personen ermöglicht, den Inhalt einer mvdXML nachzuvollziehen.

Um die mvdXML in ein grafisches Format umzuwandeln, wird diese mit Python in eine Ontologie übersetzt und anschließend ebenso mithilfe eines Python Programmes visualisiert.

II INHALTSVERZEICHNIS

III ABBILDUNGSVERZEICHNIS

IV TABELLENVERZEICHNIS

V ABKÜRZUNGSVERZEICHNIS

Abkürzung	Ausgeschrieben
MVD	Model View Definition
IFC	Industry Foundation Classes
XSD	XML-Schema-Definition
XML	Extensible Markup Language
W3C	World Wide Web Consortium
RDF	Resource Description Framework
OWL	Web Ontology Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
CAD	computer-aided Design
IcDD	Information Container for linked document delivery
UUID	universally unique identifier
HTML	Hypertext Markup Language
SPARQL	SPARQL Protocol And RDF Query Language
SHACL	Shapes Constraint Language

VI QUELLTEXTVERZEICHNIS

1 EINLEITUNG

1.1. MOTIVATION

Seit knapp 80 Jahren verändert die Digitalisierung unser aller Leben maßgeblich. Kaum ein Bereich blieb davon unberührt. Der PC hat in fast jeden deutschen Haushalt Einzug gefunden und aus der Arbeit moderner Ingenieure ist computer-aided design (CAD) nicht mehr wegzudenken. Obwohl die Geschichte des modernen Computers an der Seite des Bauingenieurs Konrad Zuse begann, spielte die Digitalisierung im Bauwesen nur eine untergeordnete Rolle. Wie in Abbildung 1 (Destatis, 2022) zu sehen, stieg die Produktivität des Bausektors im Vergleich zu anderen Branchen des produzierenden Gewerbes in den letzten Jahrzehnten nur geringfügig.

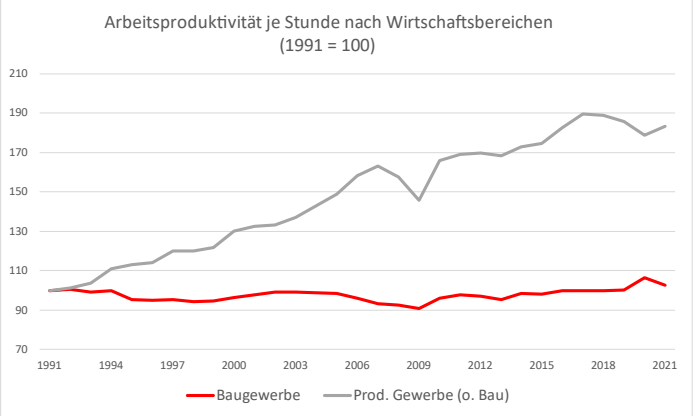

Abbildung 1 Produktivität Baugewerbe

Um dagegen anzukämpfen, wurde die BIM-Methodik entwickelt. Mithilfe dieser Methode soll vor allem Interoperabilität unterstützt und das Zusammenarbeiten verschiedener Gewerke erleichtert werden. Im Kern der BIM-Methodik stehen offene Konzepte, die zum Austausch von Informationen zwischen einzelnen Akteuren und Gewerken benutzt werden sollen. Das wichtigste Format stellt hierbei die Industry Foundation Class (IFC) dar, welche als digitales Abbild des Bauwerks in dessen Lebenszyklus zu verstehen ist. Durch die Einführung der BIM-Methodik entfernt man sich zunehmend von einer einfachen Beschreibung der geometrischen Daten eines Bauobjektes und fügt weitere, wichtige Informationen in das Datenmodell hinzu. So umfasst ein gut gepflegtes BIM-Modell einer Bahntrasse nicht nur die Länge und Position einer Schiene, sondern auch deren Material, Einbaudatum, Kosten, etc.

Durch diese Ergänzung von Informationen eröffnet sich jedoch eine vollkommen neue Herausforderung: Bei einer Vielzahl von Objekten innerhalb eines Planungsabschnittes potenziert sich die Informationsmenge auf ein Vielfaches der reinen Geometrieinformationen. Diese Informationsmenge ist in ihrer Gesamtheit zwar wichtig, birgt jedoch die Gefahr, sich darin zu verlieren. Es ist weder notwendig noch zielführend, alle Informationen stets an alle am Projekt beteiligten Parteien weiterzugeben. Um dieser Herausforderung entgegenzuwirken, wurde das Konzept der Material View Definition (MVD) entwickelt. Dieses Konzept versucht mithilfe von mvdXML-Dateien die komplexen IFC-Modelle in kleinere Submodelle zu überführen.

Die mvdXML basiert dabei auf dem XML-Schema. Dessen Aufbau wird im Kapitel 3.1 näher erläutert. Grundlegend ist jedoch dazu zu sagen, dass das XML-Schema zwar für Menschen lesbar, aber nur schwer verständlich ist.

Ziel der vorliegenden Arbeit ist es, ein Programm zu entwickeln, welches es Personen ohne IT-Hintergrund ermöglicht, die in der mvdXML enthaltenen Regeln zu verstehen und umzusetzen. Dabei wird ein besonderes Augenmerk auf den Infrastrukturbau im Bereich Bahnwesen gelegt, da hier bisher kaum Standardisierungen stattgefunden haben.

1.2. METHODIK UND VORGEHEN

Im Rahmen der Arbeit wird zuerst die offizielle mvdXML Dokumentation verarbeitet und die Struktur von mvdXML-Dateien erläutert.

Anschließend wird beschrieben welche Programme bisher in der Lage sind mvdXML-Dateien zu schreiben und zu lesen.

Es wird ein Programm geschrieben, welches in der Lage ist, mvdXML-Dateien zu importieren und diese in mvdOWL-Dateien umzuwandeln.

Im zweiten Schritt wird erläutert, wie das erstellte Programm funktioniert und welche Aspekte der Programmierung besondere Beachtung benötigen. Dabei wird auch auf Ontologien generell eingegangen.

Innerhalb der Arbeit sind alle *kursiv* geschriebenen Wörter Elemente, welche innerhalb von Programmen und externen Dateien vorkommen.

2 MODEL VIEW DEFINITIONS

2.1. GRUNDLEGENDE NUTZUNGSGEDANKEN DER MVD

BuildingSMART als Dachverband für Open BIM, gibt eine Vielzahl an Konzepten und Möglichkeiten an, eine Entität innerhalb des IFC-Standards zu repräsentieren. So ist es beispielsweise möglich, eine Schiene als einfache Linie, jedoch auch als 3D-Volumenkörper mit oder ohne weitere Attribute, darzustellen. Diese Optionen sind in der IFC angegeben. Es ist weder zielführend noch gedacht, alle Möglichkeiten gleichzeitig anzuwenden. Eher sollte der IFC-Standard als eine Bibliothek an Möglichkeiten gesehen werden, die erstellten Informationen aus einem Programm zu exportieren. Um Informationen nach ihrer Erstellung auszutauschen und in die IFC zu überführen, muss jedoch ein Export stattfinden. Es ist von buildingSMART nicht verbindlich vorgeschrieben, in welchem Datenformat dieser Export stattzufinden hat. Meistens werden die Informationen als STEP-Datei mit der Endung „.ifc“, ausgegeben. Dieses Datenformat beruht auf dem Standard zur Repräsentation von EXPRESS Datenmodellen in der ISO 10303-21:2016 (ISO, 2016).
Um in den Softwarewerkzeugen anzugeben, welche Konzepte zur Darstellung der Informationen genutzt werden, wurde die MVDs entwickelt. Sie werden im Kern der Softwarewerkzeuge implementiert, um aus der Vielzahl an möglichen Konzepten eine klare Struktur für die Weitergabe zu definieren. Sie dienen somit als Ebene der Interoperabilität (Siehe Abbildung 2).

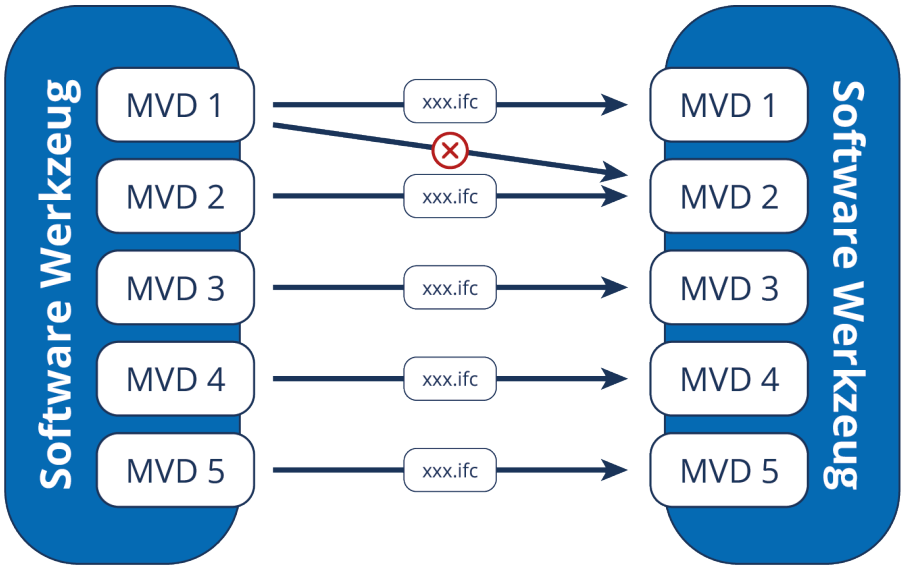

Abbildung 2 MVD als Interoperabilitätsebene

Wichtig ist hierbei, dass sowohl der Ersteller einer Datei, als auch der Empfänger derselben, auf die gleiche MVD als Basis zurückgreifen muss. Ist dies nicht der Fall, führt es zu Problemen der Interoperabilität, da gesendete Konzepte von der Empfängersoftware

eventuell nicht verstanden werden können. Diese Eigenschaft stellt die Softwareentwicklung vor eine Herausforderung, da stets auf Seiten des Senders und des Empfängers einer IFC-Datei die gleiche MVD angewendet werden muss. BuildingSMART stellt deshalb bereits einige MVDs zur Verfügung [1]. Diese Model View Definitions sind jedoch nur in einer geringen Anzahl vorhanden und bilden nicht vollumfänglich die Anforderungen des Bauwesens ab.

Grundlegend ist jedoch jeder in der Lage, eine MVD zu erstellen. Dies führt dazu, dass in verschiedenen Softwaretools eine Vielzahl an MVDs implementiert wurden, die nur eingeschränkt miteinander kompatibel sind. Um dieses Kompatibilitätsproblem zu beseitigen, wurde 2013 die mvdXML V1.0 als Datenformat von buildingSMART vorgestellt. Dieses sollte Auftraggebern die Möglichkeit geben, MVDs selbst zu spezifizieren, sodass diese an Auftragnehmer weitergereicht werden können und man auf Basis dieser Views eine einheitliche Modellstruktur erzeugt. Dementsprechend ist es notwendig, dass die benutzten Softwaretools in der Lage sind, mvdXML-Dateien zu verarbeiten. Die Struktur der mvdXML orientiert sich dabei maßgeblich am Aufbau der IFC.

In der folgenden Zeit erschloss sich eine weitere Nutzungsmöglichkeit für mvdXML-Dateien. Anhand ihrer Struktur hatte man die Möglichkeit, nicht nur Modellanforderungen vorzugeben, sondern die Datei auch als Validierungsregel zu benutzen, um eine Modellprüfung durchzuführen. Dieser Fokus wurde 2016 durch Implementierung neuer Funktionen in der mvdXML V1.1 (Chipman et al., 2016) berücksichtigt.

[1] https://technical.buildingsmart.org/standards/ifc/mvd/mvd-database/

2.2. PROGRAMME

2.2.1. MVDXML ZUM EXPORT

Bisher werden in den meisten Programmen MVDs nur proprietär eingesetzt. Vor allem innerhalb der Modellerstellungstools gibt es keine Software, welche in der Lage ist, auf Basis von mvdXML-Dateien vollumfänglich Modelle zu exportieren. Da MVDs jedoch zwingen zum Import und Export von IFC-Dateien benötigt werden, wird oft auf die standardisierten MVDs von BuildingSMART zurückgegriffen.
Wie in Abbildung 3 zu erkennen ist, unterstützt beispielsweise Autodesk Revit, als meistbenutzte Modellierungssoftware, für den Export die Model Views:

- IFC2x3 Coordination View
- IFC2x3 Concept Design BIM 2010
- IFC2x3 Basic FM Handover View
- IFC2x3 COBie 2.4 Design Deliverable
- IFC4 Reference View
- IFC4 Design Transfer View

Hierbei fällt auf, dass nicht der gesamte Umfang der MVD Database [2] von buildingSMART abgebildet ist. So fehlt etwa die „IFC4Precast“ MVD oder der „Space Boundary Addon View“. Dies ist auch nicht abhängig von der Finalisierung der einzelnen Model Views. Der Design Transfer View ist beispielsweise erst in der Entwurfsphase und dennoch schon in Autodesk Revit vorhanden.

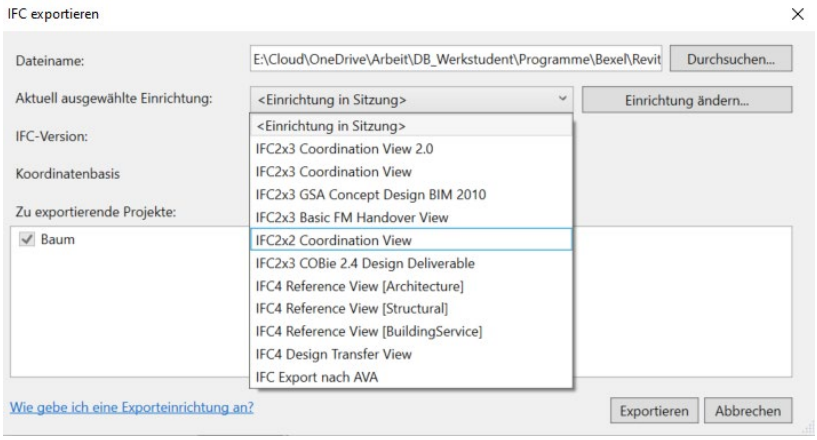

Abbildung 3 Exportmöglichkeiten Autodesk Revit

[2] https://technical.buildingsmart.org/standards/ifc/mvd/mvd-database/

2.2.2. MVDXML ZUR MODELLPRÜFUNG

Bei der Modellprüfung stützt sich SimpleBIM auf mvdXML-Dateien. Diese werden dabei jedoch nicht vollumfänglich unterstützt. SimpleBIM kann nur mit eigens vorgegebenen Concept Templates arbeiten, welche eine von SimpleBIM unterstützte universally unique identifier (UUID) besitzen. Das Programm nutzt diese UUID, um direkt die Funktionen der einzelnen Konzepte zu erkennen. Damit erlischt jedoch der Vorteil der Allgemeingültigkeit von mvdXML-Dateien.

Laut der Veröffentlichung von Weise et al., 2016, ist die Modellprüfung im Open Source Werkzeug XBIM möglich und umsetzbar. XBIM ist dabei nicht nur ein Programm, sondern es basiert auf dem .NET toolkit „xBIM toolkit“ [3]. Hierfür wurde die „mvdXML validation library“ entwickelt, welche in der Lage ist IFC-Dateien anhand von mvdXML-Dateien zu validieren. Für Endanwendende bietet das mvdXML Plugin für den XbimXplorer die selben Möglichkeiten innerhalb einer grafischen Benutzeroberfläche.

Der FZK Viewer [4] ist eine weitere Freeware, welche in der Lage ist anhand von mvdXML-Dateien eine IFC-Datei zu validieren. Er wird vom Karlsruhe Institute of Technology (KIT) entwickelt und basiert auf dem ebenfalls vom KIT entwickelten IFCExplorer. Dieser ist jedoch nicht öffentlich zugänglich. Im Rahmen der Versuche stellt sich der FZK Viewer als das einfachste und umfangreichste Werkzeug aus den genannten Möglichkeiten heraus.

Baumgärtel & Pirnbaum verfolgten einen anderen Ansatz. In deren Arbeit wird die mvdXML zuerst in eine weitere Sprache umgewandelt. Diese Sprache nennt sich IFC Query Language (ifcQL) und „ ähnelt der Sprache SQL“ (Baumgärtel & Pirnbaum, 2016). Im nächsten Schritt lässt sich mithilfe der ifcQL die IFC-Datei analysieren und den vorgegebenen Anforderungen entsprechend filtern bzw. prüfen.

Es existieren noch weitere mvdXMLChecker. Diese sind jedoch in einem Status, in dem eine Person ohne tiefe Programmierkenntnisse kaum Zugang zu ihnen findet. Deshalb werden sie in dieser Arbeit nicht weitergehend berücksichtigt.

[3] https://docs.xbim.net/
[4] https://www.iai.kit.edu/1302.php

3 MVDXML ALS DATENFORMAT

3.1. XML-DOKUMENT

Um MVDs auszutauschen, wird von buildingSMART das Format mvdXML empfohlen. Hierbei handelt es sich um eine modifizierte XML-Datei. Im Folgenden wird kurz der grundlegende Aufbau einer XML-Datei erläutert. Anschließend wird sich mit dem direkten Konzept der mvdXML auseinandergesetzt. Die folgend aufgeführten Informationen beziehen sich auf die offizielle Dokumentation der XML-Spezifikation (Bray et al., 2013) des World Wide Web Consortiums (W3C).

3.1.1. GRUNDLEGENDER AUFBAU

Die Extensible Markup Language (XML) wird als Austauschformat in verschiedensten Bereichen verwendet und ist eine Auszeichnungssprache. Durch ihre Struktur ist sie nicht nur maschinenlesbar, sondern auch für den Menschen verständlich.

Vereinfacht dargestellt kann sich eine XML als einen Baum vorstellt werden, welcher aus sogenannten „Elementen“ besteht. Jedes Element kann weitere Elemente, sowie Text enthalten. Um Elemente zu kennzeichnen, werden sogenannte „Tags“ benutzt. Dabei gibt es drei Arten von Tags (Siehe Tabelle 1).

Tabelle 1 Übersicht der einzelnen Tags

Tag	Bedeutung
<Elementname>	Tag für Beginn des Elementes
</Elementname>	Tag für Ende des Elementes
<Leer/>	Tag für leeres Element

Des Weiteren bietet das XML-Schema die Möglichkeit den Elementen jeweils Attribute zuzuweisen. Dies geschieht über einen Einschub im Start- bzw. Leer-Tag mithilfe des Syntax *Attributname = “Attributwert“.*
Um die Formbedingung des XML-Schemas zu erfüllen, muss außerdem in der ersten Ebene deklariert werden, welche Version und welches Encoding benutzt wird. Dabei kann auch auf eine XML-Schema-Definition verwiesen werden, welche den näheren Aufbau der Elemente genauer beschreibt.
Somit ergibt sich grundlegend eine Datei wie in Quelltext 1.

3.1.2. XML-SCHEMA-DEFINITION

Eine XML-Schema-Definition (XSD) wird zum Definieren von XML-Strukturen verwendet. Sie gilt als Empfehlung des W3C. XSD-Dateien haben üblicherweise die Endung „.xsd“ und sind selbst in der Struktur eines XML-Dokumentes beschrieben. Mit einer XSD lässt sich

konkret der Aufbau einer XML-Datei festlegen. Mithilfe dieser Definition ist eine Kontrolle auf inhaltliche Vollständigkeit und Korrektheit der XML-Datei erreichbar. Hierbei ist es nicht nur möglich, auf Datentypen zu prüfen, sondern auch komplexere Zusammenhänge, wie Anzahl der Elemente und Wertebereiche können vorgegeben werden.
Für mvdXML-Dateien ist eine passende XSD-Datei auf der Website von buildingSMART [5] auffindbar.

Quelltext 1 Beispiel XML-Datei

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Schienen>
        <Schiene Kilometer="3.00" Material="Stahl">
                <Dimensionen>
                        <Laenge> 5.00 </Laenge>
                        <Breite> 0.12 </Breite>
                        <Hoehe> 0.15 </Hoehe>
                </Dimensionen>
                <Strecke>4112</Strecke>
                <Schienentyp>Schwere Vignolschiene</Schienentyp>
        </Schiene>
        <Schiene Kilometer="3.00" Material="Stahl">
                <Dimensionen>
                        <Laenge> 4.50 </Laenge>
                        <Breite> 0.12 </Breite>
                        <Hoehe> 0.15 </Hoehe>
                </Dimensionen>
                <Strecke>4112</Strecke>
                <Schienentyp>Schwere Vignolschiene</Schienentyp>
        </Schiene>
</Schienen>

```

3.1.3. XML-PARSING

Da diese Arbeit in einer objektorientierten Programmiersprache geschrieben ist, muss die XML-Datei zu Beginn in ein für die Programmiersprache verständliches Format umgewandelt werden. Diese Umwandlung wird in der Fachsprache auch „Parsing“ genannt. Für Python existieren verschiedene Parsing-Werkzeuge. Im Rahmen der Arbeit wird die Lxml [6] Bibliothek benutzt. Lxml bietet im Vergleich zu anderen Optionen die Möglichkeit, während der Programmierung sehr nah an bekannten Python Strukturen zu bleiben (mehr dazu in Kapitel 5.2.1).

[5] https://standards.buildingsmart.org/MVD/RELEASE/mvdXML/v1-1/mvdXML_V1.1_add1.xsd
[6] https://lxml.de/

3.2. MVDXML ELEMENTE

Im Nachfolgenden werden kurz die wichtigsten Elemente der mvdXML-Dateistruktur, welche in Abbildung 4 zu sehen ist, beschrieben. Hierbei ist darauf zu achten, dass Subelemente innerhalb der mvdXML Datenstruktur oft gegliedert sind. Wie beispielsweise in Quelltext 2 zu sehen, werden die *AttributeRule* Elemente nicht direkt als Subelement von *ConceptTemplate* eingefügt. Stattdessen wird zuerst das Element Rules erzeugt, unter dem im Anschluss alle *AttributeRule* Elemente gesammelt sind. In Abbildung 4 wird diese Gliederung über eine Beschriftung der Zuweisungspfeile repräsentiert. Da das Gliederungselement keinerlei Informationen außerhalb der reinen Subelementzuweisung besitzt, wird es in den weiteren Beschreibungen nur angedeutet und direkt auf die Subelemente verwiesen. Des Weiteren werden nicht alle Elemente und Attribute erklärt, da eine ausführlichere Beschreibung bereits in der offiziellen Dokumentation (Chipman et al., 2016) zu finden ist und nicht alle Informationen für die Funktion des Programmes notwendig sind. Um die einzelnen Entitäten nachvollziehbarer zu gestalten, wurden Beispiele aus der Datei *bim-q-test.mvdXML* hinzugefügt. Diese Datei entstand aus einem automatischen Export der Software Bim-Q. Dabei handelt es sich um Anforderungen an IFC-Modelle der Leistungsphase 2 eines Streckenausbauprojektes.

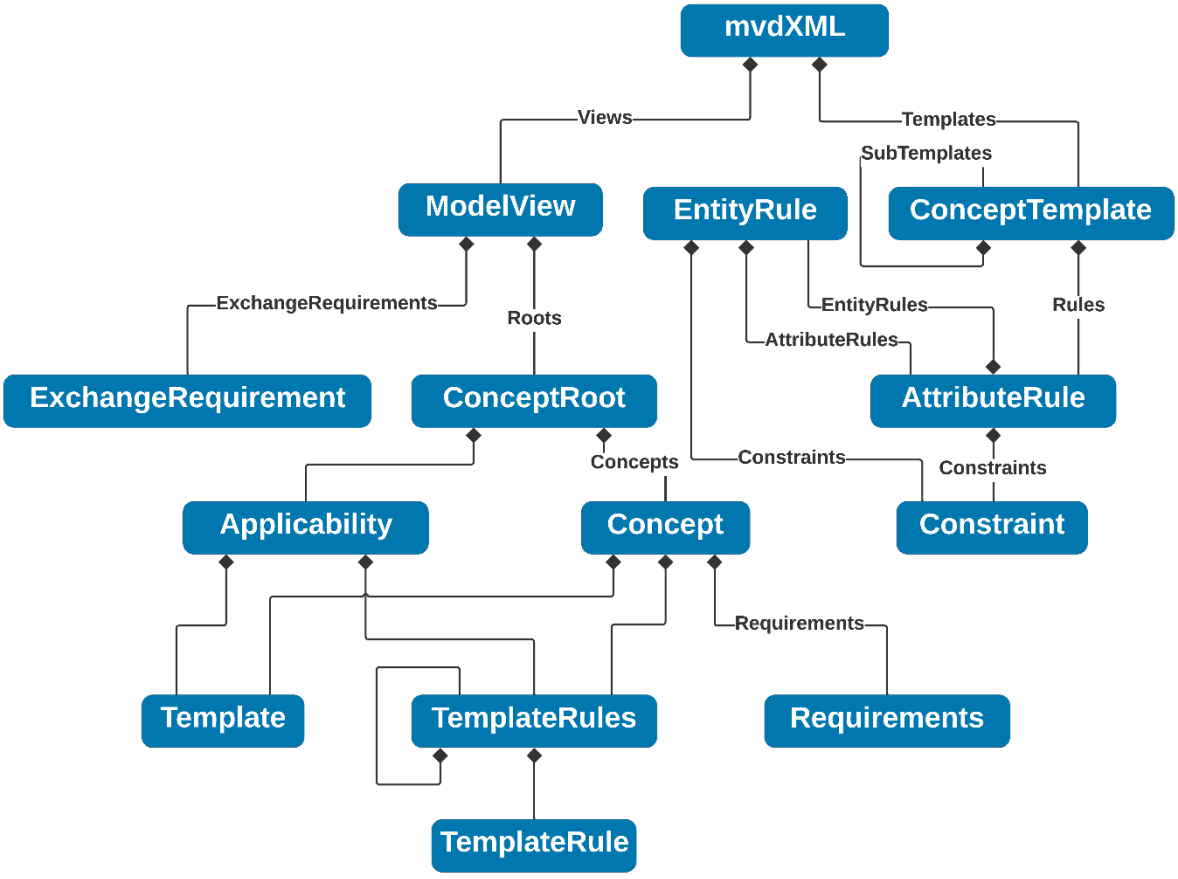

Abbildung 4 Aufbau mvdXML

Quelltext 2 Schachtelung mvdXML

```
<ConceptTemplate uuid="00000000-0000-0000-0001-000000000001" name="ProductConceptTemplate"
applicableSchema="IFC4" applicableEntity="IfcProduct">
   <Definitions> …
   <Rules>
     <AttributeRule  RuleID="Name" AttributeName= "Name"/>
     <AttributeRule  RuleID="Description" AttributeName= "Description"/>
     <AttributeRule  RuleID="Tag" AttributeName= "Tag"/>
     <AttributeRule  RuleID="ContainedInStructure" AttributeName= "ContainedInStructure"/>
     <AttributeRule  RuleID="Decomposes" AttributeName= "Decomposes"/>
…
```

3.2.1. MVDXML

Quelltext 3 Beispiel mvdXML Element

```
<mvdXML
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://buildingsmart-tech.org/mvd/XML/1.1"
   uuid="5065a878-c99e-43d5-b0c2-862deb053f13" name="" status="sample"
   xsi:schemaLocation=http://www.buildingsmart-tech.org/mvd/XML/1.1/mvdXML_V1.1_add1.xsd>
```

Das *mvdXML* Element ist die erste Ebene in der XML-Datenstruktur. Pro mvdXML-Datei darf nur ein *mvdXML* Element existieren. Die Subelemente des Elementes sind das *Template* Element, welches aus *ConceptTemplate* Elementen besteht und das *Views* Element, welche aus *ModelView* Elementen besteht. Des Weiteren besitzt das *mvdXML* Element die Möglichkeit, Identitätsattribute zu besitzen (Siehe Kapitel 3.2.11). Neben den Informationen, welche in der offiziellen mvdXML V1.1 Dokumentation angegeben werden, kann das Element auch die Attribute, welche in Kapitel 3.1.1 für die erste Ebene angegeben sind, beinhalten.

Tabelle 2 mvdXML Element

Sub Klassen	ConceptTemplate, ModelView
Attribute	Identitätsattribute
Anzahl	1

3.2.2. CONCEPTTEMPLATE

Quelltext 4 Beispiel ConceptTemplate Element

```
<ConceptTemplate
uuid="00000000-0000-0000-0001-000000000001"
name="ProductConceptTemplate"
applicableSchema="IFC4"
applicableEntity="IfcProduct">
```

Dieses Element ist im Grunde eine wiederverwendbare Vorlage, um ModelViews zu definieren. Dabei kann das *ConceptTemplate* Element als Subelemente sowohl Attributregeln *(AttributeRule)* über das *Rules* Element beinhalten, als auch weitere *ConceptTemplates* über das *SubTemplates* Element. Um das *ConceptTemplate* näher zu definieren, kann außerdem das *Definitions* Subelement benutzt werden. Neben den Identitätsattributen kann auch ein anwendbares Schema definiert werden, sowie eine anwendbare Entität. Die anwendbare Entität bezieht sich herbei auf die IFC Datenstruktur und gibt vor, dass das betreffende ConceptTemplate für alle Entitäten und deren Erben verfügbar ist. Das *ConceptTemplate* wird im Rahmen der Visualisierung als Startpunkt der Knoten gesehen. Dabei wird die angegebene Entität jedoch vom *ConceptRoot* Element vorgegeben.

Tabelle 3 ConceptTemplate Element

Sub Klassen	ConceptTemplate, Definitions, AttributeRule
Attribute	Identitätsattribute, applicableSchema, applicableEntity, isPartial
Anzahl	[0:?]

3.2.3. ATTRIBUTERULE

Quelltext 5 Beispiel AttributeRule Element

```
5431 <AttributeRule
     RuleID="Value"
     AttributeName= "NominalValue">
```

Das *AttributeRule* Element verweist mithilfe des *AttributeName* Attributes auf ein spezifisches Attribut seines Elternelementes. Das Elternelement kann entweder ein *ConceptTemplate* oder eine *EntityRule* sein. Falls durch ein *Concept* auf eine entsprechende *AttributeRule* verwiesen werden soll, so muss diese eine *RuleID* besitzen. Diese wird innerhalb der Parameterdefinition einer *TemplateRule* verwendet. In der Regel verweisen *TemplateRule* Elemente auf *AttributeRule* Elemente, da damit eine Zuweisung von Attributen ermöglicht wird. Subelemente des *AttributRule* Elementes können *EntityRule* Elemente oder *Constraint* Elemente sein. Da *Constraint* Elemente jedoch selten benutz werden, wurden sie in dieser Arbeit nicht näher betrachtet.

Tabelle 4 AttributeRule Element

Sub Klassen	Constraints, EntityRule
Attribute	AttributeName, RuleID, Description
Anzahl	[0:?]

3.2.4. ENTITYRULE

Quelltext 6 Beispiel EntityRule Element

```
5433 <EntityRule
     EntityName="IfcValue">
```

Das *EntityRule* Element wird benutzt, um Entitäten innerhalb des IFC Standards darzustellen. Dabei gibt das *EntityName* Attribut den Namen der entsprechenden Entität an. Mithilfe des *RuleID* Attributes kann, genau wie bei der *AttributeRule*, auf die *EntityRule* verwiesen werden. Die *EntityRule* gibt somit immer die Entität an, auf die das Elternelement, also die *AttributeRule*, verweist. Da es in verschiedenen *ConceptTemplates* eventuell zu Dopplungen von EntityRule-AttributeRule-Kombinationen kommen kann, ist es möglich, mithilfe des *Reference* Elementes auf andere *ConceptTemplates* zu verweisen, um deren Konzepte zu benutzen. Diesem Verweis kann mithilfe des *IdPrefix* Attributes ein Prefix angehängt werden, welcher allen folgenden *RuleIDs* hinzugefügt wird.

Tabelle 5 EntityRule Element

Sub Klassen	Constraints, AttributeRule, References
Attribute	EntityName, RuleID, Description
Anzahl	[0:?]

3.2.5. MODELVIEW

Quelltext 7 Beispiel ModelView Element

```
6692  <ModelView
      uuid="00000291-3493-0000-0000-000000000000"
      name="Beispiel Projektarbeit"
      applicableSchema="IFC4">
```

Der *ModelView* wird, wie der Name bereits erkennen lässt, dafür verwendet die Model View Definition zu spezifizieren. Der View muss sich dabei zwingend auf ein bestimmtes IFC Schema beziehen. Dabei stehen vier Möglichkeiten für das Attribut *applicableSchema* zur Auswahl:

- IFC2X_FINAL
- IFC2X2_FINAL
- IFC2X3
- IFC4

Neben den *ConceptRoot* Elementen, welche die wichtigen *TemplateRule* Elemente enthalten, verweist das *ModelView* Element auch möglicherweise auf *ExchangeRequirements*. Diese wird im Rahmen der Projektarbeit allerdings nicht weiter erklärt, da sie hauptsächlich eine beschreibende Rolle einnimmt und nur indirekt Einfluss auf das Prüfverhalten einer mvdXML-Datei nimmt.

Tabelle 6 ModelView Element

Sub Klassen	ConceptRoot, ExchangeRequirement, BaseView, Definitions
Attribute	Identitätsattribute, applicableSchema
Anzahl	[0:?]

3.2.6. CONCEPTROOT

Quelltext 8 Beispiel ConceptRoot Element

```
6709  <ConceptRoot
      uuid="00000291-3493-3493-0000-000000808724"
      name="1-MA OLA Mast"
      applicableRootEntity="IfcBuildingElementProxy">
```

Das *ConceptRoot* Element ist das fundamentale Element der MVD. Innerhalb eines *ConceptRoot* Elementes sind die wichtigsten Informationen enthalten, welche genutzt werden, um einen ModelView zu spezifizieren. Jedes *ConceptRoot* Element hat als Subelemente ein *Applicability* Element und null bis mehrere *Concept* Elemente. Anhand des *applicableRootEntity* Attributes wird angegeben, auf welche Entitäten des IFC Schemas sich die Subelemente beziehen. Im Quelltext 7 wird sich beispielsweise auf *IfcBuildingElementProxy* Entitäten bezogen. Anhand des Attribut *name*, lässt sich bereits erahnen, dass es sich hierbei um einen Oberleitungsmast handeln soll. Da in der IFC4 Spezifikation noch keine eigene Entität für Oberleitungsmasten existiert, muss dieser als ein Proxy Element übergeben werden. Um dieses Element von anderen Proxy Elementen zu unterscheiden ist es nun notwendig, weiterer Attribute für eine klare Objekttypenzuweisung herzustellen. Die gängige Praxis ist hierbei ein benutzerdefiniertes PropertySet zu erstellen und das Attribut „Objekttyp“ hinzuzufügen. Im betrachteten Beispiel wird das PropertySet „Bestandsdaten“ genannt. Innerhalb des ConceptRoot Elementes ist jedoch kein Verweis auf die Objekttypenzuweisung zu finden. Das liegt daran, dass diese Verweise innerhalb des *Applicability* Elementes abgebildet werden.

Tabelle 7 ConceptRoot Element

Sub Klassen	Applicability, Concept, Definitions
Attribute	Identitätsattribute, applicableRootEntity
Anzahl	[0:?]

3.2.7. APPLICABILITY

Quelltext 9 Beispiel Applicability Element

```
6715 <Applicability>
```

Wie in Kapitel 3.2.6 bereits kurz erklärt, wird das *Applicability* Element benutzt, um die, von den Regeln betroffenen, Entität näher zu spezifizieren. Das Element ist vom Aufbau her dem *Concept* Element sehr ähnlich. Mithilfe der *TemplateRule* Elemente werden die betroffenen Entitäten anhand ihrer Eigenschaften definiert. Das *Applicability* Element besitzt keine eigenen Attribute, sondern beinhaltet nur Subelemente. Um die Vollständigkeit der mvdXML-Datei zu gewährleisten, muss das *Applicability* Element immer existieren.

Tabelle 8 Applicability Element

Sub Klassen	TemplateRules, TemplateRule, Definitions
Attribute	
Anzahl	1

3.2.8. CONCEPT

Quelltext 10 Beispiel Concept Element

```
6795 <Concept
         uuid="00000291-3493-3493-0000-000000808790"
         name=" G-03 Kilometrierung: 4-GEM Gemeinsame Eigenschaften: 1-MA OLA Mast:
              01-1-OLA Oberleitungsanlagen: BMOD Bahnmodell "
         code=" ">
```

Dieses Element wird benutzt, um für eine, mithilfe der *ConceptRoot* und *Applicability* deklarierte Entität, Regeln aufzustellen. Das *Concept* bildet hierbei die grundlegende Verbindung zwischen ModelView und *ConceptTemplate*, denn durch das *Template* Element wird mithilfe einer spezifischen *UUID* auf ein *ConceptTemplate* verwiesen. Nur auf dieses *ConceptTemplate* und dessen Subelemente werden sich im Anschluss die einzelnen *TemplateRule* Elemente beziehen.

Tabelle 9 Concept Element

Sub Klassen	TemplateRules, TemplateRule, Definitions, Template
Attribute	Identitätsattribute, BaseConcept, Override, Requirements
Anzahl	[0:?]

3.2.9. TEMPLATERULES

Quelltext 11 Beispiel TemplateRules/TemplateRule Element

```
<TemplateRules operator="and">
   <TemplateRules operator=„or“>
      <TemplateRules operator=„or“>
         <TemplateRule Parameters= "Set[Value]='Bestandsobjekt' AND
                                    PropertyName[Value]='Vermessungszeitraum' AND
                                    Value[Value]='1. Quartal 2020'"/>
         <TemplateRules operator="and">
            <TemplateRules operator="nor">
               <TemplateRule Parameters= "Set[Value]='Bestandsobjekt' AND
                                          PropertyName[Value]='Vermessungszeitraum'"/>
            </TemplateRules>
         <TemplateRules operator=„or“>
            <TemplateRule Parameters= "T_Set[Value]='Bestandsobjekt' AND
                                       T_PropertyName[Value]='Vermessungszeitraum' AND
                                       T_Value[Value]='1. Quartal 2020'"/>
            </TemplateRules>
         </TemplateRules>
      </TemplateRules>
   </TemplateRules>
</TemplateRules>>
```

Mithilfe des *TemplateRules* Elementes ist es möglich, mehrere *TemplateRule* oder *TemplateRules* Elemente logisch zu verknüpfen. Dabei stehen die Operatoren AND, OR, NOT, NAND, NOR, XOR, NXOR zur Verfügung. Dies geschieht über das *operator* Attribut. Dieses Element besitzt, wie das *ConceptTemplate* Element auch, die Möglichkeit ein Subelement des Eigenen Typen zu besitzen. Dadurch hat man die Möglichkeit, komplexe Logikstrukturen durch Verschachtelung zu erzeugen. Wie in Quelltext 9 jedoch bereits zu erkennen ist, wird die Datei bei einer komplexen Struktur schnell unübersichtlich.

Tabelle 10 Templates Element

Sub Klassen	TemplateRule, Definitions
Attribute	
Anzahl	[0:?]

3.2.10. TEMPLATERULE

Das *TemplateRule* Element bildet den Kern jeder mvdXML-Datei. Das Element besitzt, außer einem optionale *Description* Element, keine weiteren Subelemente. Die Hauptfunktion liegt im *Parameter* Attribut. Mithilfe des Attributes lassen sich klare Regeln aufstellen und logisch miteinander verknüpfen. Dies bildet die Basis für die Modellvalidierung. Der Wert des Parameter Attributes, folgt einer vorgegebenen Syntax (Quelltext 10).

Parameter Syntax

Der grundlegende Aufbau der Parameter Syntax wird in Chipman et al. (2016) folgend definiert:

Quelltext 12 Parameter Syntax

```
Parameter = expression
expression
    :boolean_expression;
boolean_expression
    : boolean_term (logical_interconnection Boolean_term)*
boolean_term
    : parameter ( metric )? | metric ) operator ( value | parameter ( metric )? ) ) |
    ( LPAREN boolean_expression RPAREN );
```

Die Variablen *parameter*, *metric*, *operator* und *value* werden innerhalb der Dokumentation detaillierter aufgeschlüsselt. Primär von Bedeutung ist die Möglichkeit, *boolean_terms* beliebig oft mithilfe von logischen Verknüpfungen (*logical_interconnection*) zu verbinden. Die komplexe Struktur der *boolean_terms* kann in der Realität oft auf diese Syntax heruntergebrochen werden:

Quelltext 13 vereinfachung boolean_term

```
1 parameter metric operator value
```

Betrachtet man beispielsweise die *TemplateRule* aus Quelltext 9

```
1 <TemplateRule Parameters=
2 "Set[Value]='Bestandsobjekt' AND
3 PropertyName[Value]= 'Vermessungszeitraum' AND
4 Value[Value]='1. Quartal 2020'"/>
```

So lässt sich diese, wie in Abbildung 20 gezeigt, dekonstruieren. Die *logical_interconnection* ist bei allen *boolean_terms* eine UND-Verknüpfung.
Mithilfe von Quelltext 11 lassen sich nun die boolean_terms aufschlüsseln.

Für den ersten *boolean_term* ergeben sich somit folgende Variablen:

- parameter: “Set“
- metric: “[Value]“
- operator: “=“
- value: „‘Bestandsobjekt‘

Um diesen *boolean_term* vollständig zu verstehen, muss nun noch die *parameter* Variable aufgeschlüsselt werden. Deren Wert ist eine *RuleID*. Wie bereits unter 3.2.8 beschrieben ist, verweist das *Concept*, dessen Kind-Element die *TemplateRule* ist, auf ein *ConceptTemplate*. Innerhalb dieses ConceptTemplates muss nun eine *AttributeRule* oder eine *EntityRule* existieren, welche die entsprechende *RuleID* besitzt. Die Abfolge der Elemente, die zum betreffenden Element führt, ergibt schlussendlich die Definition der geforderten IFC Spezifikation. Damit erschließt sich für den ersten *boolean_term* eine Spezifikation welche in Abbildung 5 zu erkennen ist. Dabei stellen die grauen Kacheln Entitäten dar und die blauen Kacheln symbolisieren Attribute der Entitäten. In diesem Beispiel muss der Name eines Property Sets dem Wert „Bestandsdaten“ entsprechen.

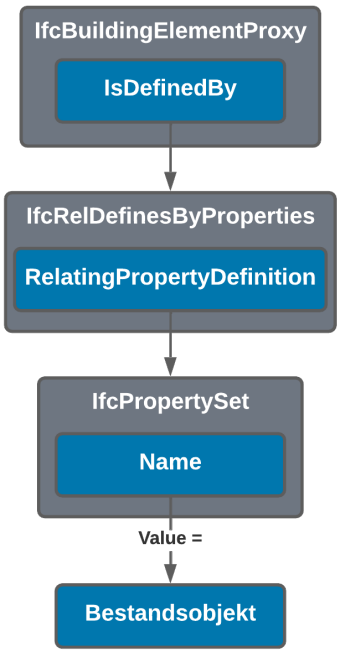

Abbildung 5 boolean_term 1

Dieses Vorgehen ist für den zweiten *boolean_term* und den dritten *boolean_term* identisch. *Boolean_term 2* ergibt somit, dass der Name des betreffenden Properties ‚Vermessungszeitraum‘ lautet. Aus *boolean_term 3* lässt sich schlussendlich erschließen, dass der Vermessungszeitraum des betreffenden Objektes dem Wert ‚1. Quartal 2020‘ entsprechen soll.

Zusammenfassend ergeben die Regeln die Informationen welche in Abbildung 6 dargestellt werden.

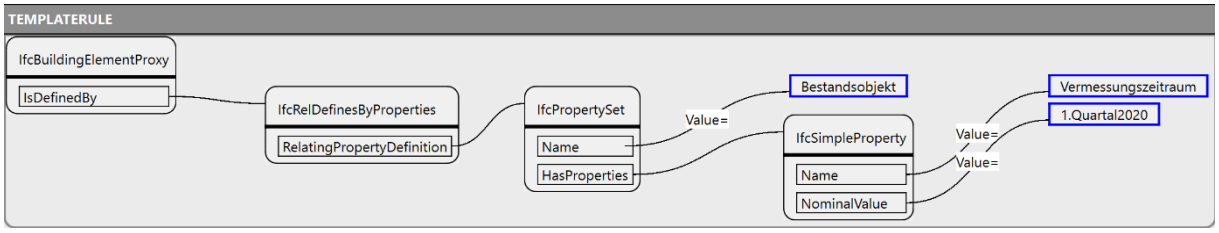

Abbildung 6 Beispiel Visualisierung

3.2.11. IDENTITÄTSATTRIBUTE

Identitätsattribute sind jene Attribute, welche in Tabelle 11 aufgeführt sind. Dabei ist zu beachten, dass nicht alle Attribute verpflichtend existieren müssen. Identitätsattribute werden benutzt, um einzelnen Elementen innerhalb einer mvdXML eine Eindeutigkeit zu verleihen und weiter Informationen zur Identifizierbarkeit anzufügen. Um die Identifizierbarkeit zu gewährleisten, muss jedes Element, welches Identitätsattribute zugewiesen hat, eine eindeutige UUID besitzen. Eine UUID ist eine Textkette mit einer Länge von 36 Zeichen.

Tabelle 11 Identitätsattribute

Attribut	Typ	Optional
UUID	uuid	Nein
Name	String	Nein
Code	String	Ja
Version	String	Ja
Status	String	Ja
Author	String	Ja
Owner	String	Ja
Copyright	String	Ja

3.3. VOR & NACHTEILE MVDXML

3.3.1. VORTEILE

Mithilfe von mvdXML-Dateien ist es möglich, komplexe Strukturen und Anforderungen darzustellen. Dies führt zu einer Vereinheitlichung von Schnittstellen zwischen unterschiedlichen Softwaretools. Nur durch eine exakte Vorgabe von exportierten Informationen ist es möglich, diese auch korrekt zu importieren. Deshalb spielt die MVD eine wichtige Rolle in der Interoperabilität zwischen einzelnen Programmen.

Mithilfe des mvdXML Formates wurde außerdem ein vielfältiges Werkzeug geschaffen, welches es ermöglicht, deutlich komplexere Prüf- und Exportregeln zu erstellen als in vergleichbaren Formaten.

Die mvdXML-Spezifikation ist gut dokumentiert und offen zugänglich. Es fallen keine Lizenzgebühren oder weitere Kosten bei der Verwendung von mvdXML-Dateien an.

3.3.2. NACHTEILE

Die vielfältige Einsetzbarkeit von mvdXML-Dateien führt jedoch auch zu einer erhöhten Komplexität. Diese wirkt im ersten Moment oft überfordernd oder abschreckend. Möglicherweise wurde auch deshalb das mvdXML Format bisher nur unzureichend in Programmen implementiert. Oft bieten Softwarewerkzeuge nur die Möglichkeit, einzelnen MVDs als Exportspezifikation zu benutzen, ohne jedoch den Nutzenden die Möglichkeit zu geben, eigene Exportanforderungen mithilfe von mvdXML-Dateien zu definieren. Im Rahmen der zukünftigen Entwicklung des IFC Standards wird auch deshalb das Konzept der mvdXML-Dateien durch Information Delivery Specifications (IDS) ersetzt (buildingSMART International Ltd., 2022).

4 WEB ONTOLOGY LANGUAGE

Da innerhalb einer mvdXML-Datei intensiv mit Beziehungen zwischen den einzelnen Elementen gearbeitet wird, wurde sich im Rahmen der Projektarbeit dafür entschieden, dass es sinnvoll ist, die mvdXML-Datei in eine Ontologie umzuwandeln.

Ontologien sollen es Maschinen ermöglichen, Beziehungen zwischen Entitäten nutzbar zu machen und mithilfe von Logik deren Informationen zu nutzen. Vor allem im Rahmen des Semantic Web (Siehe Kapitel 4.1) und der enormen Informationsmenge des Internets ist es notwendig, einheitliche und verarbeitbare Strukturen zu erschaffen. Dabei folgen Ontologien einer klaren Struktur, was im Kontrast zum oft genutzten Machine Learning steht (Siehe Jean-Baptiste, 2021). Das Wissen, welches innerhalb von Ontologien gespeichert wird, setzt sich aus zwei großen Teilbereichen zusammen.
Jede Entität innerhalb einer Ontologie kann inhärente Informationen besitzen. Durch die Verbindung einzelnen Entitäten, entstehen Relationen. Durch das entstehende Netz aus Entitäten und Relationen, lassen sich implizite Informationen in explizite Informationen umwandeln (Siehe Kapitel 4.4). Es entsteht somit ein Wissenszuwachs.
Die meistverwendete Methode, um Ontologien darzustellen ist die Web Ontology Language (OWL). Deshalb wurde sich im Rahmen der Projektarbeit dafür entschieden, mvdXML-Dateien in OWL-Dateien mit der Endung „.mvdOWL“ umzuwandeln.

4.1. SEMANTIC WEB

Das „klassische“ Web besteht aus Hypertext Markup Language (HTML) Dokumenten. HTML ist eine Auszeichnungssprache (Siehe W3C, 2022), entwickelt durch das World Wide Web Consortium (W3C). Inhalte eines HTML Dokumentes haben keine klar vorgegebene Struktur. Sie können vom Erstellenden beliebig angeordnet werden. Somit ist es für einen Computer nicht ersichtlich, welchen Sinn ein gewisser Inhalt (z.B. Text oder Bild) in einem HTML Dokument erfüllt. In einem HTML Dokument wird beispielsweise nicht deutlich, ob es sich bei einem Bild um ein Foto einer Schiene handelt oder um eine genaue Montageanweisung derselben. Angenommen, es handelt sich dabei um ein Foto einer Schiene, so ist weiterhin nicht erschließbar, ob die Schiene bereits in einem vorherigen Textabschnitt beschrieben wurde oder eventuell bereits weitere Fotos der Schiene in dem Dokument eingebettet sind. Das ist für die maschinelle Verarbeitung der existierenden Daten ein Hindernis und führt vor allem bei großen Datenmengen zu Problemen bei deren Auswertung.

Mithilfe des Semantic Webs und Ontologien soll dafür eine Lösung geschaffen werden. Den Bildern soll nun beispielsweise eine klare Struktur zugeordnet werden. Sie besitzen damit die Information über ihren Inhalt, sowie Verknüpfungen zu Textelementen, welche sie referenzieren und weitere Bilder, die dieselbe Schiene abbilden.

Um diese Metadaten zu beschreiben, wird vom der W3C das Resource Description Framework (RDF) empfohlen. Um Ontologien darzustellen, nutzt die OWL die Methoden, welche von RDFs zur Verfügung gestellt werden.

4.2. RESOURCE DESCRIPTION FRAMEWORK

Das RDF wird benutzt, um logische Aussagen über Dinge zu treffen. Dinge werden hierbei in der Fachsprache als „Ressourcen“ bezeichnen. Jede Aussage im RDF besteht aus drei Elementen:

- Subjekt
- Prädikat
- Objekt

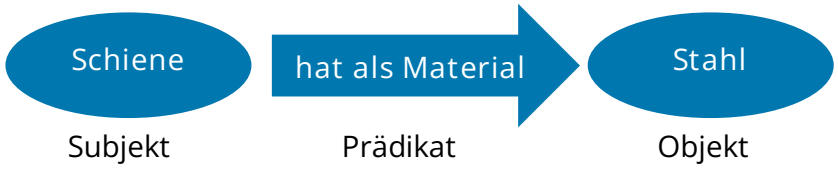

Abbildung 7 RDF Tripel

Wie in Abbildung 7 dargestellt, wird das Subjekt durch das Objektes näher beschrieben. Zur Verbindung wird das Prädikat genutzt. Es entsteht ein sogenanntes „Tripel“. Subjekt und Prädikat müssen stets Ressourcen sein. Das Objekt besitzt, neben den Ressourcen, auch die Möglichkeit ein Literal zu sein. Hierbei handelt es sich um eine direkte Angabe von Informationen, beispielsweise Zahlen, Daten oder Text. Mehrere Tripel ergeben ein RDF-Modell.

RDF-Modelle werden meist als XML-Datei mit der Endung „.rdfxml“ gespeichert. Außerdem wurde hierfür vom W3C auch die Sprache „Turtle“ entworfen. Zur Abfrage von RDF-Modell entstand die graphenbasierte „SPARQL Protocol And RDF Query Language“ (SPARQL), welche an SQL angelehnt ist (Siehe W3C SPARQL Working Group, 2013). Um komplexe Anforderungen an RDF-Dateien zu stellen, wird zudem die Shapes Constraint Language (SHACL) vom W3C vorgegeben (Siehe Knublauch & Kontokostas, 2017).

Damit Ressourcen global eindeutig referenziert werden können, wird ein eindeutiger Bezeichner, der Uniform Resource Identifier (URI), genutzt. Der aus dem „klassischen“ Web bekannte Uniform Resource Locator (URL) ist eine besondere Art der URI. Im Gegensatz zu einer URL kann, muss eine URI aber nicht unbedingt in einem Netzwerk erreichbar sein.

4.3. AUFBAU

OWL und der 2009 entstandene Nachfolger OWL2 (W3C OWL Working Group, 2012) nutzen das RDF, um die Struktur des Wissens zu beschreiben. Durch die Nutzung des RDF ist es möglich, die OWL als Teil des Semantic Webs zu benutzen. Durch eine fest vorgegebene Syntax, können verschiedenen OWL-Ontologien miteinander kombiniert werden. Dadurch ist es möglich, größere Zusammenhänge zu erschaffen und einen Wissenszuwachs herzustellen.

4.3.1. KLASSEN

Individuelle Entitäten werden im RDF als Ressourcen bezeichnet. Innerhalb der OWL spricht man anschließend von einzelnen Instanzen bzw. Objekten. Eine Sammlung von Objekten ist eine Klasse. Klassen können wiederum weitere Instanzen einer Klasse beinhalten und zu keiner, einer oder mehreren Klassen zugeordnet sein. Wie auch bei objektorientierten Programmiersprachen können Klassen als Subklassen anderer Klassen existieren und somit deren Eigenschaften erben. In der OWL ist jede Klasse eine Subklasse von „owl:Thing“ und hat als Subklasse „owl:Nothing“. In Abbildung 8 wird beispielsweise die Klassifizierung einer Schiene vorgenommen. Diese ist Teil des Oberbaus und übernimmt dementsprechend dessen Methoden und Informationen.

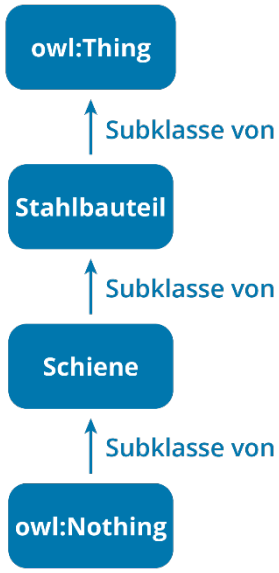

Abbildung 8 Subklassen OWL

Die Zuweisung einzelner Instanzen zu einer Klasse kann entweder explizit oder implizit geschehen, wobei bei einer expliziten Zuweisung eine sogenannte ClassAssertion genutzt wird. Das linguistische Äquivalent wäre am Beispiel der Abbildung 8 „Das Objekt ‚Schiene 12.54‘ ist eine Schiene“. Mehr zu impliziten und expliziten Informationen in Kapitel 4.4.

4.3.2. EIGENSCHAFTEN

Eigenschaften sind gerichtete Informationen, welche Klassen beschreiben. Wenn einer Klasse eine Eigenschaft zugewiesen ist, so muss diese für alle Instanzen der Klasse zutreffen. Eigenschaften können von mehreren Klassen geteilt werden. Deshalb besteht die Möglichkeit eine Eigenschaft einer Basis (Domain) und einem Ziel (Range) zuzuordnen. Eine Eigenschaft kann entweder eine gerichtete Zuweisung zu einem Wert oder zu einer anderen Instanz sein. Dabei können Eigenschaften logische Verhaltensweisen annehmen. So können diese einfach, symmetrisch, invers, funktional oder transitiv sein, welche folgen erläutert werden.

Einfache Eigenschaft

Die einfache Eigenschaft E beschreibt eine Verbindung zwischen dem Objekt X und einem Objekt Y oder dem Objekt X und einer Information Y. Falls es sich um eine Verbindung zwischen Objekt und Objekt handelt, so spricht man von einem „ObjectProperty“, welches durch die Vererbung der owl:ObjectProperty Klasse signalisiert wird. Sollte eine Verbindung zwischen Objekt und Information stattfinden, so spricht man von einem DatatypeProperty, gekennzeichnet durch die Vererbung der owl:DatatypeProperty Klasse.

Abbildung 9 Einfache Eigenschaft

Symmetrische Eigenschaft

Eine symmetrische Eigenschaft E bedeutet, dass falls die Eigenschaft E von X zu Y gilt, auch die Eigenschaft E von Y zu X gilt. Ist beispielsweise ein Oberleitungsmast neben einer Schiene, so ist die Schiene zwingend auch neben dem Oberleitungsmasten. Symmetrische Eigenschaften werden als solche definiert, indem man sie als Instanz der Klasse owl:SymmetricProperty definiert.

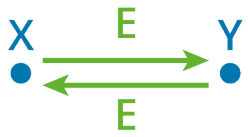

Abbildung 10 Symmetrische Eigenschaften

Inverse Eigenschaften

Da Eigenschaften stets gerichtet sind, ist es möglich der Eigenschaft eine inverse Eigenschaft zuzuordnen, welche entgegen gerichtet ist. Falls die Eigenschaft E1 von X zu Y gilt, so kann die inverse Eigenschaft E2 von Y zu X gebildet werden. Im Gegensatz zu symmetrischen Eigenschaften ist es bei inversen Eigenschaften möglich, verschieden Eigenschaften zu verknüpfen. Wird beispielsweise das ObjectProperty E1 „Schiene liegt auf Schwelle“ definiert, so kann als Inverse Eigenschaft E2 „Schwelle liegt unter Schiene“ hinzugefügt werden. Wird nun innerhalb einer Ontologie E1 hinzugefügt, erschließt sich E2 automatisch und muss nicht händisch ergänzt werden.

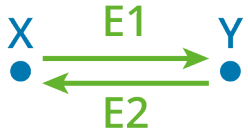

Abbildung 11 Inverse Eigenschaften

Funktionale Eigenschaften

Eine funktionale Eigenschaft beschreibt, dass die Eigenschaft, die ein Objekt besitzt, eindeutig belegt sein muss. Die vorherigen Eigenschaften können aus multiplen Tupeln bestehen. So kann, wie im Beispiel der inversen Eigenschaft, eine Schiene auf mehreren Schwellen gleichzeitig liegen. Eine funktionale Eigenschaft verbietet eine solche Zuordnung. Soll beispielsweise die Betonklasse einer Schwelle angegeben werden, so ist dies eine funktionale Eigenschaft, da ein Beton nie mehrere Betonklassen gleichzeitig besitzen kann.

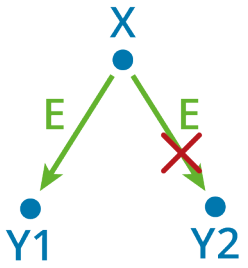

Abbildung 12 Funktionale Eigenschaften

Einer funktionalen Eigenschaft kann auch eine invers funktionale Eigenschaft zugeordnet werden. Dies kommt in den Anwendungen des Bauwesens sehr häufig vor. Das Beispiel aus der inversen Eigenschaft, kann auch als invers funktional angenommen werden. Voraussetzung ist hierfür, wie eine Schiene definiert ist. Bei einer allgemeinen Betrachtung wird davon ausgegangen, dass Schienen meist paarweise auftreten. Im Kontext der Infrastrukturplanung wird deshalb diese paarweise Anordnung direkt als Schiene bezeichnet und nicht weiter aufgeteilt. Geht man nun davon aus, dass eine Schiene als der einzelne Metallkörper definiert ist, so ist das Beispiel „Schwelle liegt unter Schiene“ nicht funktional, da jede Schwelle unter zwei Schienen liegen würde. Wird eine Schiene jedoch als ein Paar an Metallkörpern definiert, so ist das Beispiel funktional und dementsprechend die Eigenschaft „Schiene liegt auf Schwelle“ invers funktional. Eine Schiene kann somit auf mehreren Schwellen liegen, eine Schwelle kann sich aber jeweils nur unter einer Schiene befinden.

Transitive Eigenschaft

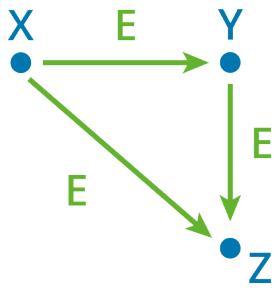

Abbildung 13 transitive Eigenschaften

Angenommen die Eigenschaft E ist als Transitiv deklariert. Bildet man nun eine Instanz der Eigenschaft E aus den Objekten X und Y, sowie eine Instanz der Eigenschaft E aus den Objekten Y und Z, so lässt sich indirekt erschließen, dass X und Z ebenfalls diese Eigenschaft teilen. Wenn beispielsweise ein Projekt in mehrere Planungsabschnitte aufgeteilt ist, und es in den Planungsabschnitten einzelne Objekte (z.B. Masten) gibt, so kann die transitive Eigenschaft „istTeilVon“ erstellt werden. Aus der Eigenschaft ‚Schiene istTeilVon Projektabschnitt‘ und ‚Projektabschnitt istTeilVon Projekt‘, lässt sich somit erschließen, dass ‚Schiene istTeilVon Projekt‘ ebenfalls zutrifft. Transitive Eigenschaften werden als solche definiert, indem man sie sie als Instanz der Klasse owl:TransitiveProperty definiert.

4.3.3. OPERATOREN

Durch die OWL sind mehrere Operationen zwischen Klassen einer Ontologie und teilweise kompletten Ontologien möglich. So unterstützt die OWL Operatoren der Mengentheorie, wie Schnittmengen, Vereinigungsmengen und Komplemente. Außerdem können für Klassen einer Ontologie Kardinalitäten, Disjuktionen und Äquivalenzen vorgegeben werden. Mithilfe der Operatoren lassen sich komplexe Zusammenhänge darstellen, welche zum Gewinn von impliziten Informationen (Siehe 4.4) genutzt werden können.

4.3.4. TEILSPRACHEN

Es gibt mehrere Sprachen, welche als OWL gekennzeichnet sind. Diese Teilsprachen kennzeichnen sich jeweils durch eine Erweiterung der Syntax des Vorgängers aus. Wie in Abbildung 14 ersichtlich ist die einfachste Teilsprache OWL Lite. Durch eine Erweiterung des Funktionsumfanges von OWL Lite, wird OWL DL erhalten. Schlussendlich existiert OWL Full. Jede korrekte Ontologie in OWL Lite ist automatisch auch eine korrekte Ontologie in OWL DL und jede korrekte Ontologie in OWL DL ist automatisch auch eine korrekte Ontologie in OWL Full. Das Inverse dieser Aussagen ist allerdings nicht korrekt.

Abbildung 14 Teilsprachen OWL

4.4. IMPLIZITE UND EXPLIZITE INFORMATIONEN

Ein wichtiger Aspekt von Ontologien ist das sogenannte „Reasoning“. Dabei kontrolliert ein sogenannter „Reasoner“, ob die Ontologie in sich konsistent ist; Ob beispielsweise die Klassen, welche mit einer Eigenschaft verbunden sind, in deren Range und Domain wiederzufinden sind. Außerdem entsteht die Möglichkeit, aus angegebenen (expliziten) Informationen Zusammenhänge zu erschließen (implizite Informationen). Im Rahmen der Projektarbeit wird dies hauptsächlich genutzt, um inverse Relationen zu erhalten, ohne diese erneut händisch anzugeben. Wird beispielsweise dem Programm eine Concept übergeben, welches mithilfe des Template-Elementes auf ein ConceptTemplate verweist, so weiß das ConceptTemplate auch automatisch, dass es von diesem spezifischen Concept referenziert wird.

Als Reasoner wurde sich im Rahmen der Projektarbeit für „HermiT“ entschieden, dieser ist sowohl in der benutzten Python-bibliothek OWLReady2 (Siehe Kapitel 5.2.2) als auch in Protégé, einem bekannten Ontologie Programm, vorhanden.

4.5. IFCOWL

Da das Semantic Web Vorteile gegenüber klassischen Datenstrukturen bietet, gibt es auch die Bestrebung innerhalb des BIM-Feldes, den IFC-Standard im Semantic Web nutzbar zu machen. Hierfür wurde die ifcOWL entwickelt (Siehe Pauwels & Terkaj, 2016). Dabei handelt es sich um eine Repräsentation der IFC-Spezifikation in der OWL. Durch die ifcOWL ist es möglich, die Semantic Web Technologien mit dem IFC Standard zu verbinden. Der Nutzen ist hierbei offensichtlich: Durch die Möglichkeit, Relationen zwischen Objekten zu spezifizieren, ist es nicht nur möglich, den kompletten IFC-Standard abzubilden, sondern auch weitere Dokumente, wie beispielsweise Produktdatenblätter oder Zeitplanungselemente, einem Objekt anzuhängen. Das Zusammenspiel zwischen IFC-Standard und Semantic Web findet auch bei der „Information Container for linked document delivery (Icdd)“, beschrieben in der ISO 21597-1:2020 (ISO, 2020), Anwendung. Hierbei handeltet es sich um ein Konzept, welches den Datentransfer zwischen Projektbeteiligten unterstützen soll. Aktuell werden hauptsächlich einzelne Dokumente zwischen Teilnehmenden eines Projektes geteilt, dadurch gehen oft Zusammenhänge zwischen den Dokumenten verloren. Die Icdd ermöglichen es, mithilfe von Ontologien, weitere Dokumente und Informationen mit Entitäten einer IFC-Datei zu verknüpfen. In unterstützenden Anwendungen, wird hierfür oft die IFC-Datei, welche im STEP Format vorliegt, in eine ifcOWL umgewandelt.

5 PROGRAMM

Das erstellte Programm soll, wie bereits erklärt, in der Lage sein eine mvdXML-Datei in eine Ontologie zu transformieren. Mithilfe der Ontologie wird eine grafische Benutzeroberfläche erstellt, welche die MVD visualisiert. Das Programm wurde mit der objektorientierten Sprach Python geschrieben. Für die Arbeit mit Ontologien wird in der Regel Java verwendet. Jedoch bietet Python den Vorteil, dass es leichter zu lernen ist und eine größere Community besitzt. Zwar ist Python eine relativ langsame Programmiersprache (Vgl. Srinath, 2017) besitzt dabei aber die Möglichkeit, bei Rechenoperationen mithilfe der Bibliothek „Cython“ [7] auf die Leistung von nativem C und C++ Code zurückzugreifen. Im folgenden Kapitel werden die wichtigsten Funktionen, Bibliotheken und Klassen erklärt. Das Programm ist auf GitHub [8] abrufbar, und auf dem beigelegten USB-Stick gespeichert.

5.1. AUFBAU

Im Rahmen der Projektarbeit wurde ein sogenannten „Python-Package“ erstellt. Das bedeutet, dass es sich um eine Sammlung von mehreren Dateien, welche importiert werden können, handelt. Grundlegend besteht das erstellte Package aus drei Modulen. Module sind Python-Dateien, welche Funktionen, Variablen und Klassen besitzen, die von Nutzern importiert werden können. Ein Fokus liegt hier vor allem darauf, wiederverwendbare Methoden und Objekte zu erzeugen. Um die mvdXML-Datei in eine Ontologie umzuwandeln, wurde das Module *core.py* erstellt. Darin sind alle Methoden und Objekte vorhanden, die die Arbeit mit Ontologien betrifft. Genutzt wird dieses Modul hauptsächlich durch das *visualization.py* Modul. Darin befinden sich alle Objekte und Methoden zur Visualisierung der verarbeiteten mvdXML. Um wiederkehrende Werte innerhalb der grafischen Umgebung nicht direkt in den Code einzubetten, wurde das Modul *constants.py* erstellt, darin sind hauptsächlich numerische Variablen gespeichert, wie der Abstand zwischen Objekten oder vordefinierte Namen von Objekten.

Neben dem Python-Package befindet mit sich in den digitalen Anlagen (Siehe Anlage 1) mit der Datei *mvd2OWL_precompiled.zip* auch eine vorkompilierte Version des Programmes.

[7] https://cython.org/
[8] https://github.com/c-mellueh/MVD2OWL

5.2. BENUTZTE BIBLIOTHEKEN

Um den Programmieraufwand zu reduzieren und Redundanzen zu anderen Projekten gering zu halten, wurden mehrere Open Source Bibliotheken benutzt. Dabei wurden primär die Bibliotheken LXML, OWLREADY2 und PYSIDE benutzt.

5.2.1. LXML

LXML ist eine Python Bibliothek, die „die Einfachheit der nativen Python API besitzt, fast komplett kompatibel, jedoch deutlich überlegen zur ElementTree API ist“ (Richter, 2012). Da sich LXML stark an der ElementTree API orientiert, sind viele elementare Methoden identisch in der Anwendung. LXML wandelt alle Elemente einer XML-Datei in Python-Objekte um. Angenommen, Quelltext 1 befindet sich in der Datei „quelltext_1.xml“., so würde der einfachste Befehl wie in Quelltext 12 lauten.

Quelltext 14 Parse Quelltext 1

```
1  from lxml import etree,objectify
2
3  et = etree.parse(“quelltext_1.xml”)
```

Das Objekt *et* bildet nun die komplette Baumstruktur des XML-Dokumentes ab. Es ist möglich, auf die einzelnen Elemente zuzugreifen. In der Regel wird zuerst das oberste Element der Baumstruktur aktiviert (Siehe Quelltext 13). Der Vorteil von LXML ist eine Funktionsweise, die sehr direkt an die Python-interne Struktur angelegt ist. So erhält man z.B. mit der Funktion *root.getchildren()* eine Liste aller Subelemente des entsprechenden XML-Elementes (hier *root*). Auf die Tags, Attribute und Inhalte der einzelnen Elemente kann ebenso leicht zugegriffen werden (Siehe Quelltext 14). Dabei ist wichtig, dass die Attribute als *Dictionary* Objekt übergeben werden. Mithilfe der Funktion *root.find(name)* ist möglich Kind Elemente des root Objektes anhand ihres Tags zu erhalten.

Quelltext 15 getroot()

```
from lxml import etree

et = etree.parse("quellcode_1.xml")
root = et.getroot()
print(root)
```

```
Output:
          <Element Schienen at 0x2776d8ddc80>
          Process finished with exit code 0
```

Quelltext 16 Attribute und Tags

```
schiene_1:etree._Element = root.getchildren()[0]

print(schiene_1.tag)
print(schiene_1.attrib)
print(schiene_1.text)
```

```
Output:
          Schiene
          {'Kilometer': '3.00', 'Material': 'Stahl'}
```

5.2.2. OWLREADY2

Mithilfe des Owlready2 Packages hat man die Möglichkeit, OWL2.0 Ontologien in Python zu importieren, bearbeiten und erstellen. Im Rahmen der Arbeit wird Owlready2 ausschließlich dafür benutzt, neue Ontologien zu erstellen. Dabei orientiert sich Owlready2, genau wie LXML, sehr direkt an den Python-internen Methoden. Wie eine grundlegende Ontologie erzeugt wird, ist in Quelltext 15 zu sehen. Dabei sind bereits mehrere Konzepte zu beachten. Um Ontologie-Klassen (*Schiene*, *Schwelle*, *Zug*, *Person*) zu erstellen, muss ihnen das Objekt *Thing* vererbt werden. Dieses beinhaltet die Basisinformationen, die eine Klasse benötigt. Zur Erstellung von Relationen, wird entweder die *ObjectProperty* Klasse oder die *DataProperty* Klasse vererbt. Das ist davon abhängig, ob die Relation zwischen einem Objekt und einem anderen Objekt besteht, oder zwischen einem Objekt und einer Information. Funktionale Relationen werden zusätzlich über die Vererbung der Klasse *FunctionalProperty* bzw. *InverseFunctionalProperty* gekennzeichnet. Um angeben zu können, welche Objekte in der Lage sind, gewisse Relationen zu besitzen, wird die Variable *domain* zu den Relationen hinzugefügt. Mithilfe der *range* Variable wird definiert, auf welche Objekte sich die Relation beziehen darf. Wenn es mehrere Objekte gibt, die gleichnamige Relationen besitzen können, so muss dies mit der *Or()* Funktion gekennzeichnet werden. Da in einer mvdXML-Datei Relationen und Beziehungen zwischen Objekten stets eindeutig sind, wird im Rahmen des Projektes ausschließlich mit funktionalen Relationen gearbeitet. Dies bietet, neben der informativen Korrektheit, außerdem den Vorteil, dass Owlready2 in der Programmierung dadurch einfacher zu benutzen ist. Grund hierfür ist das Verhalten der Bibliothek bei der Anforderung von Relationen. Ist eine Relation funktional, so lässt sich das referenzierte Element direkt als Variable übergeben (Siehe Quelltext 16). Bei einer nichtfunktionalen Relation werden alle Elemente, die referenziert werden, in einer Liste gespeichert. Zur Erweiterung bzw. Änderung der Liste müssen somit immer die Listenoperatoren von Python genutzt werden (Siehe Quelltext 17). Da dem Inversen einer funktionalen Relation mehrere Objekte zugeordnet sein können, ist der entsprechende Werte wieder eine Liste. Dieses Verhalten bietet die Möglichkeit, effizient mit der Datenstruktur zu arbeiten, da einerseits ohne unnötigen Programmieraufwand Werte übergeben werden können, andererseits schnell über vorhandene Informationen iteriert werden kann.

Quelltext 17 Initialisierung Ontologie

```
from owlready2 import *

onto = get_ontology("http://schiene/onto.owl")

with onto:
    class ElementMitName(Thing):

        def __init__(self, name):
            super().__init__()
            self.hat_name = name

    class Schiene(ElementMitName):
        pass

    class Schwelle(ElementMitName):
        pass

    class Zug(ElementMitName):
        pass

    class Person(ElementMitName):
        pass

    class liegt_unter(ObjectProperty):
        domain = [Schwelle]
        range = [Schiene]

    class liegt_auf(ObjectProperty):
        domain = [Schiene]
        range = [Schwelle]
        inverse_property = liegt_unter

    class wird_befahren_von(ObjectProperty):
        domain = [Schiene]
        range = [Zug]

    class faehrt_auf(ObjectProperty):
        domain = [Zug]
        range = [Schiene]
        inverse_property = wird_befahren_von

    class beinhaltet_personen(ObjectProperty, InverseFunctionalProperty):
        domain = [Zug]
        range = [Person]

    class sitzt_in(ObjectProperty, FunctionalProperty):
        domain = [Person]
        range = [Zug]
        inverse_property = beinhaltet_personen

    class hat_name(DataProperty, FunctionalProperty):
        domain =  [Or([Schiene , Schwelle , Zug, Person])]
        range = [str]
```

Quelltext 18 Beispiel funktionale Relation

```
zug_1 = Zug("RE 16 (74500)")
person_1 = Person("Max Mustermann")
person_1.sitzt_in = zug_1

print(zug_1.beinhaltet_personen)
print(person_1.sitzt_in)
```

```
Output:
          onto.zug1
          [onto.person1]
```

Quelltext 19 Beispiel nichtfunktionale Relation

```
schwelle_1 = Schwelle("Betonschwelle_km24.1")
schwelle_2 = Schwelle("Betonschwelle_km24.1")
schiene_1 = Schiene("Schiene_km_24")
schiene_1.liegt_auf.append(schwelle_1)
schiene_1.liegt_auf.append(schwelle_2)

print(schwelle_1.liegt_unter)
print(schiene_1.liegt_auf)
```

```
Output:
          [onto.schiene1]
          [onto.schwelle1,onto.schwelle2]
```

Owlready2 bietet nicht nur die Möglichkeit, in Python Ontologien zu erstellen und damit zu arbeiten, sondern auch deren Export wird unterstützt. Mit dem Befehl *onto.save(file)* wird die erstellte Ontologie automatisch in eine .rdfxml-Datei umgewandelt, welche zur weiteren Verarbeitung nutzbar ist. Mithilfe des sync_reasoner() befehls kann auch das Reasoning aktiviert werden. Innerhalb der Projektarbeit wurde jedoch hauptsächlich Protégé benutzt, um nachträglich die Ontologie zu kontrollieren und das Reasoning durchzuführen.

5.2.3. PYSIDE

Durch PySide besteht die Möglichkeit, auf das Framework Qt zuzugreifen. Dabei handelt es sich um ein Werkzeug zum Erstellen von grafischen Benutzeroberflächen. Qt ist in C++ geschrieben. Durch PySide wird nun Qt innerhalb von Python nutzbar gemacht. Dabei wird sich nahe an der Python-Methodik orientiert. Es gibt jedoch auch einige Abweichungen. Das erstellte Programm ist für Qt6 geschrieben.

Qt beinhaltet verschiedene Frameworks um Objekte darzustellen und diese zu nutzen. Als Basis der meisten Userinterfaces gelten sogenannte „Widgets“. Diese sind hauptsächlich für den Input und Output von Informationen zuständig und statisch innerhalb des Fensters angeordnet. Bei Widgets handelt es sich beispielsweise um:

- QLabel
- QButton
- QTreeView
- QCheckBox

Widgets können in sogenannte Layouts eingefügt werden, um ihnen eine Struktur zu gegeben. Sollen beispielsweise mehrere Widgets vertikal gestapelt werden, so ist es sinnvoll, diese in ein QVBoxLayout einzufügen.

Im Rahmen der Projektarbeit wird maßgeblich mit dem Graphics View Framework gearbeitet. Hierbei handelt es sich um die Klasse QGraphicsView, welche selbst ein Widget ist. Sie ermöglicht es, eine Vielzahl an 2D Objekten darzustellen. Die QGraphicsView wird folgend als View bezeichnet. Der View selbst ist nicht das Objekt, welches die 2D Objekte beinhaltet. Die 2D Objekte sind in einer QGraphicsScene gespeichert, welche folgend als Szene bezeichnet wird. Der View wird benutzt, um eine Szene darzustellen. Dabei ermöglicht der View das Zoomen und Transformieren einer Szene. Die Szene umfasst, wie bereits beschrieben, alle 2D Objekte. Dabei handelt es sich beispielsweise um:

- QGraphicsRectItem
- QGraphicsEllipseItem
- QGraphicsTextItem

Allerdings ist es möglich, nicht nur klassische 2D Objekte wie Rechtecke oder Kreise in eine Szene einzubetten. Über das QGraphicsProxyWidget ist es möglich, einer Szene Widgets hinzuzufügen. Somit ergibt sich eine Möglichkeit, Widgets als 2D Objekte in einer Szene einzubetten. Dadurch wird eine Verknüpfung von 2D Objekten und Widgets ermöglicht.

5.3. UMWANDLUNG IN ONTOLOGIE

Alle Klassen und Funktionen die notwendig sind, um die mvdXML-Datei in eine Ontologie umzuwandeln, sind in der Datei „core.py“ vorhanden. Um die Transformation zu starten, muss ein MvdXML Objekt erstellt werden, dem der Dateipfad zur mvdXML-Datei übergeben wird. Im Rahmen der Instanziierung gibt es die Möglichkeit, die angegebene mvdXML-Datei direkt gegen eine XSD-Datei zu validieren. Der Ablauf des Importes ist für alle mvdXML-Objekte ähnlich. Zur Initialisierung wird den Objekten mithilfe der LXML Bibliothek ihr entsprechendes XML-Objekt übergeben. Objekte mit Identitätsattributen (Siehe Kapitel 3.2.11) erhalten über die Funktion *import_identity_data* die entsprechend vorhandenen Identitätsattribute. Anschließend werden mithilfe der *import_sub_elements* und *import_elements* Funktionen Subelemente des betroffenen Elementes importiert. Durch die *import_attribute* Funktion lassen sich schließlich auch die vorhandenen Attribute den entsprechenden Objekten zuweisen.

Einfache Attribuierungen und Subelemente sind somit schnell und einfach als Python Objekte importiert und als Ontologie übergeben. Um Verknüpfungen zwischen einzelnen Elementen herzustellen, die außerhalb einer reinen hierarchischen Struktur existieren, müssen allerdings weitere Funktionen implementiert werden. Im Rahmen der Projektarbeit wird nicht auf jede Funktion im vollen Umfang eingegangen. Alle Funktionen sind mithilfe von Dokumentations-Strings im Code näher beschrieben. Auf die wichtigsten Funktionen wird in Kapitel 5.4 eingegangen.

Nicht alle Variablen die innerhalb von *core.py* benutzt werden, sind innerhalb der erstellten Ontologie abgebildet. Die erstellte Ontologie beinhaltet nur Informationen, die bereits in der ursprünglichen mvdXML-Datei vorhanden waren. Hilfsvariablen bleiben nur im Python-Code vorhanden und werden final nicht in die Ontologie überführt. Eine Abweichung der Struktur zwischen Ontologie und mvdXML-Datei findet nur an wenigen Stellen statt. Beispielsweise ist, wie in 3.2.10 beschrieben, *Parameters* eigentlich nur ein Attribut des *Template* Elementes. Durch die tiefergreifende Logik in der Syntax dieses Attributes, wurde sich im Rahmen der Projektarbeit dazu entschieden eine *Parameter* Klasse in der Ontologie einzuführen. Dadurch wird die Ontologie übersichtlicher und es entsteht die Möglichkeit, den Parameters String in kleinere Partitionen aufzuteilen (siehe 5.4.1), sodass dieser nicht immer verarbeitet werden muss.

5.4. CORE.PY

5.4.1. DECONSTRUCT_PARAMETER()

Wie bereits in 3.2.10 erklärt, bildet die TemplateRule das Kernelement einer normalen mvdXML-Datei. Hierbei ist das Attribut *Parameters* dafür zuständig, die aufgestellten Regeln abzubilden. Bevor Parameters verarbeitet werden kann, wird es jedoch, im Kontrast zur mvdXML, als eigenes Element der Ontologie hinzugefügt und mit der entsprechenden TemplateRule verknüpft. Da der übergebene *Parameters* String, welcher als *self.has_for_plain_text* im *Parameter* Element gespeichert ist, aus mehreren logisch trennbaren Teilen besteht, wurde sich dazu entschieden, mithilfe der deconstruct_parameter() Funktion den String in folgende Attribute aufzuteilen:

- parameter
- metric
- operator
- value

Diese Aufteilung erschließt sich bereits aus Quelltext 10. Daraus ist auch zu erkennen, dass der Parametersyntax deutlich von der einfachen Form

Quelltext 20 vereinfachter Parametersyntax

```
parameter ( metric ) operator ( value )
```

abweichen kann, dies findet in der Regel jedoch sehr selten statt. In den betrachteten mvdXML-Dateien, welche aus BIM-Q stammen oder händisch erstellt wurden, wird bisher nur mit der vereinfachten Form (Quelltext 18) gearbeitet. Deshalb wurde sich im Rahmen der Projektarbeit dazu entschieden nur diese Form zu verarbeiten und bei anderen Formen eine Fehlermeldung auszugeben.

In Quelltext 19 ist zu erkennen, dass der Parameters String, mithilfe der Regular Expression in Zeile 4, in seine Bestandteile aufgeteilt wird. Die einzelnen Bestandteile werden teilweise mit weiteren Funktionen verarbeitet, um sie in ihre nativen Datenformate umzuwandeln (`self.import_value`) oder auf die interne Korrektheit zu überprüfen (`self.import_metric`).

Bisher wird davon ausgegangen, dass alle *boolean_terms* mit einer AND-Verknüpfung verbunden sind. Diese ist die häufigste Form der Verknüpfung. Deshalb wurde sich dazu entschieden, vorerst nur diese Art der Verknüpfung zuzulassen.

Quelltext 21 deconstruct_parameter()

```
def deconstruct_parameter(self):

          helper = self.has_for_plain_text.replace(" ", "")
          pattern = re.compile("(.+)\\[(\D+)\\](=|>=|<=|>|<)(.+)")
          text_helper = re.search(pattern, text_helper)
          if text_helper is not None:
                    self.parameter = text_helper.group(1)
                    self.metric = self.import_metric(text_helper.group(2))
                    self.operator = text_helper.group(3)
                    self.value = self.import_value(text_helper.group(4))

else:
                    raise AttributeError("parameter is not correctly defined: " +
                                         str(self.has_for_plain_text))
          return (self.parameter, self.metric, self.value)

```

5.4.2. GET_LINKED_RULES()

Die get_linked_rules() Funktion ist die Basis des erstellten Programmes. Sie verknüpft die *TemplateRule* mit den referenzierten *AttributeRule* und *EntityRule* Elementen. Um dies zu erreichen, ist es wichtig, dem Schema der mvdXML zu folgen. Eine einfache Suche nach der RuleID innerhalb der XML-Datei würde eventuell zu mehreren Treffern und falschpositiven Verknüpfungen führen. Vor allem durch die Möglichkeiten von IdPrefixes (Siehe 3.2.4 EntityRule) ist es enorm wichtig, der Baumstruktur der mvdXML-Datei zu folgen.

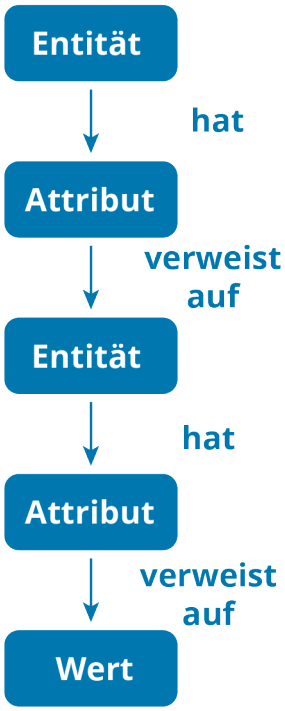

Abbildung 15 Beispielpfad

Die Funktion ist Teil der TemplateRule Klasse. Wie in Abbildung 21 zu sehen, ermittelt die TemplateRule mithilfe der get_parent() Funktion, unter welchem Concept die TemplateRule in der mvdXML-Datei untergeordnet ist. Durch das Concept lässt sich bestimmen, welches ConceptTemplate für das Referenzieren der RuleID zuständig ist. Mithilfe der Ermittlung der ConceptRoot und deren Attribut applicableRootEntity lässt sich nun bereits auch erschließen, welcher Element der IFC-Spezifikation von der Regel betroffen ist. Außerdem lässt sich über das ConceptRoot Element auch auf die Applicability zugreifen, welche die betroffenen Elemente detaillierter spezifiziert. Anschließend wird über alle Parameter iteriert, welche die TemplateRule besitzt. Mithilfe der Funktion find_rule_id() (Siehe 5.4.3), wird die referenzierte AttributeRule bzw. EntityRule gefunden und der Weg von ConceptTemplate bis zum entsprechenden Element als Variable path übergeben. Anschließend wird dem Verlauf an erster Stelle die ConceptRoot hinzugefügt, da diese immer als Startpunkt in der IFC-Spezifikation zu sehen ist. Der finale Pfad wird danach dem Parameter-Objekt übergeben. Ein Beispielpfad ist in Abbildung 15 dargestellt. Durch diese Funktion beinhaltet nach Initiierung des Programmes jedes Parameter Objekt das Wissen über den Pfad, welcher zur referenzierten Attribute- oder EntityRule führt. Dieser Pfad ist es auch, welcher grundlegend genutzt wird, um im Anschluss die mvdXML-Datei zu visualisieren. Der Pfad, welcher als Liste übergeben wird, beinhaltet nicht nur die Namen der einzelnen Objekte, sondern die Objekte selbst. Somit ist es im Anschluss problemlos möglich, auf alle Attribute und Verknüpfungen dieser Objekte zuzugreifen und diese auszuwerten. Dementsprechend existiert durch diese Funktion kein Informationsverlust.

5.4.3. FIND_RULE_ID()

Die Funktion existiert in den Klassen ConceptTemplate, AttributeRule und Entity Rule. Grund dafür ist, dass mithilfe dieser drei Klassen, wie in 3.2.10 beschrieben, der Ablauf definiert wird, welche die geforderte IFC-Struktur beschreibt. Wie in Abbildung 21 zu erkennen, ähneln sich die einzelnen Abläufe der Funktionen sehr stark. Grundlage ist stets eine Iteration über die untergeordneten Elemente des betroffenen Elementes. Im Fall von ConceptTemplates ist es sowohl möglich über weitere SubTemplates als auch über AttributeRules zu iterieren. AttributeRules ermöglichen es nur über EntityRules zu iterieren. Wie in 3.2.4 beschrieben, besitzen EntityRules nicht nur die Möglichkeit AttributeRules als Subelemente zu besitzen, sondern es ist auch möglich, auf ein weiteres ConceptTemplate zu verweisen. Bei jeder Iteration und bei jedem Verweis auf ein anderes Element wird wieder die find_rule_id() Funktion in diesem Element gestartet. Ziel dieser Vorgehensweise ist es, über alle möglichen Objekte zu iterieren, die eine RuleID referenzieren. Dabei wird kein möglicher Pfad ausgelassen und die Reihenfolge stringent beachtet. Bei jedem Aufruf der Funktion wird die zu suchende RuleID (*ruleid*), der bisher abgeschrittene Pfad (*path*) und bereits hinzugefügte Prefixes (*prefix*) der Funktion übergeben. Innerhalb der Attribute und EntiteRule wird anschließend überprüft, ob die eigene RuleID, in Kombination mit dem angegebenen Prefix, der geforderten RuleID entspricht. Ist dies der Fall, so wird der bereits abgeschrittene Pfad als Ausgabe der Funktion zurückgegeben. Sollten die RuleID nicht übereinstimmen, so wird, wie bereits beschrieben, über die Subelemente iteriert. Wenn deren Ausgabe ein Pfad ist, so wird dieser zurückgegeben. Ist dies nicht der Fall, so wird nach Abschluss aller Iterationen Null zurückgegeben. Durch dieses Verfahren stoppt die Iteration sofort, wenn das betroffene Objekt mit der passenden RuleID gefunden wird.

5.5. VISUALIZATION.PY

Mithilfe des Programmes Qt Designer, lässt sich das Grundgerüst des Userinterfaces erstellen. Dafür wird das erstellte Fenster aufgeteilt (Siehe Abbildung 16). Auf der linken Seite des Fensters existiert eine Spalte für einen TreeView, welcher im Anschluss eine Übersicht über alle Concepts beinhaltet. Auf der rechten Seite wird ein GraphicsView erzeugt, welcher benutzt wird, um im Anschluss die IFC-Struktur zu visualisieren. Zum Start des Programmes sind beide Bereiche noch leer. Die Implementierung der grundlegenden Elemente geschieht mithilfe der Funktion setup_ui() innerhalb der Klasse UiMainWindow.

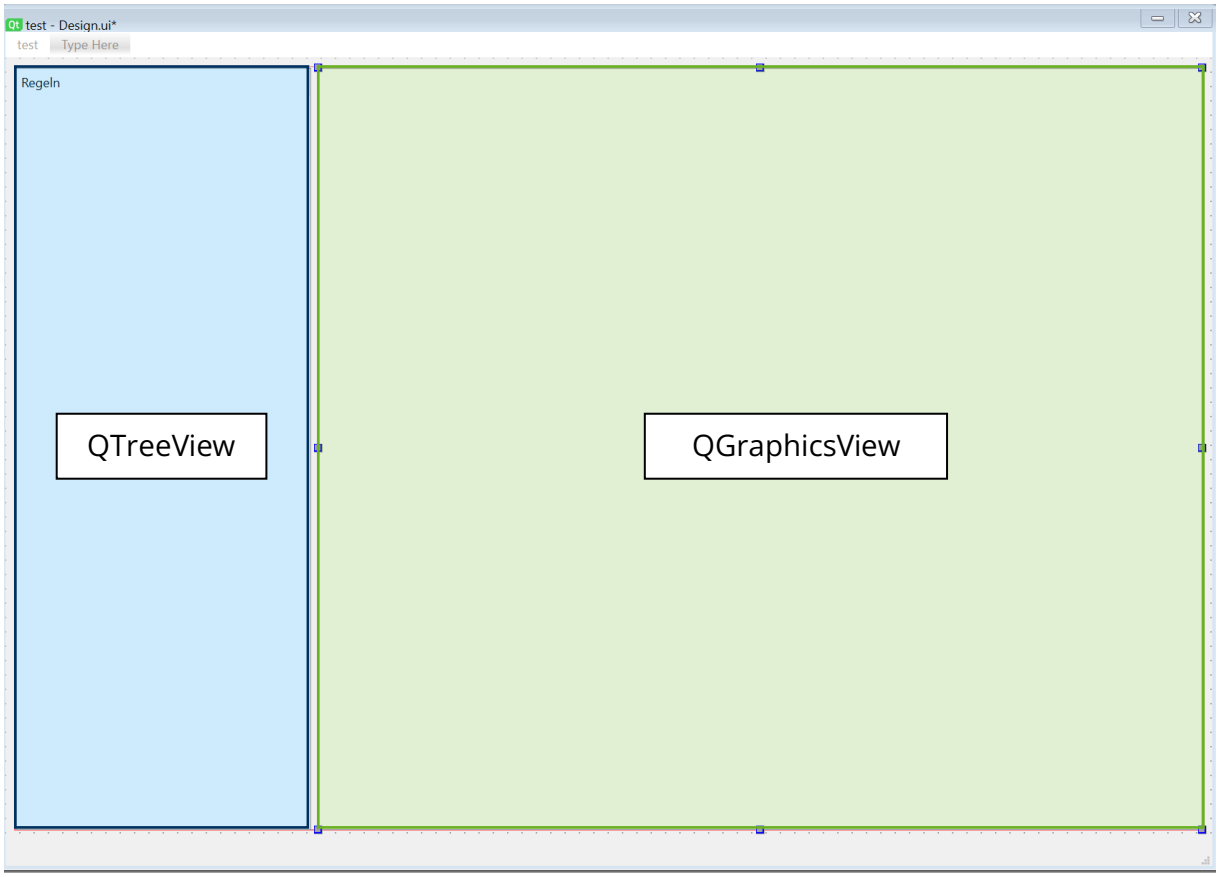

Abbildung 16 Aufteilung Fenster

Nachdem das Userinterface aufgebaut wurde, wird der TreeView gefüllt. Hierzu wird über alle Instanzen der Klasse ConceptRoot iteriert. Diese werden in der ersten Ebene des TreeViews mit deren Namen hinzugefügt. Sollten sie keinen Namen besitzen, so wird der Name über die Konstante UNDEFINED_ROOT_NAME ergänzt. Alle Concepts, welche ein Teil des entsprechenden ConceptRoot-Elementes sind, werden innerhalb der Iteration direkt in der nächsten Ebene des TreeViews abgelegt. Allen Elementen im Treeview werden die zugehörigen Ontologie-instanzen als Daten angehängt, damit keine Informationen verloren gehen. Falls auf ein Element des TreeViews geklickt wird, startet die Funktion on_tree_clicked(), welche die betreffende TemplateRule der MVD visualisiert.

5.5.1. TEMPLATERULEGRAPHICSVIEW

Jede TemplateRule ist innerhalb des Programmes ein eigener View der Klasse TemplateRuleGraphicsView. Jeder View besitzt einen TitleBlock (Siehe Abbildung 17). Dies hat den Vorteil, dass eine TemplateRule in eine andere Szene als Widget eingefügt werden kann. Wie in Abbildung 17 zu erkennen, befinden sich folgende Objekte innerhalb eines TemplateRuleGraphicsViews:

- EntityRepresentations
- Connections
- Labelrepresentations

Diese Objekte können beliebig oft eingefügt werden und dienen zur grafischen Repräsentation des erstellten Pfades aus Kapitel 5.4.2.

Die EntityRepresentation ist ein sogenanntes QFrame Widget, welches modifiziert wurde, um Entitäten der IFC Spezifikation sinnvoll darzustellen. Ein QFrame ist eine Ansammlung von Widgets, welche über ein Layout gruppiert und gezielt formatiert werden kann. Die EntityRepresentation hat immer ein vertikales Layout. Es besteht aus einem Titelelement, welches den Namen der Entität beinhalten. Gefolgt wird dies von einer horizontalen Linie. Anschließend kann eine beliebige Menge an QLabel Elementen eingefügt werden, die die einzelnen Attribute eine Entität innerhalb der IFC Spezifikation repräsentieren.

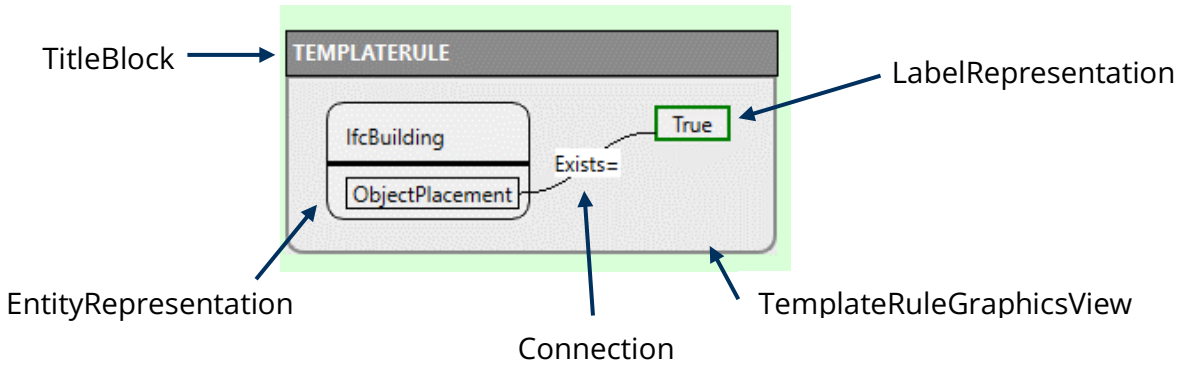

Abbildung 17 Aufbau TemplateRule

Die Connection Klasse ist eine Kombination eines QPainterPath Objektes und einem QGraphicsTexItem. Das QPainterPath Objekt wird genutzt, um eine Linie zwischen zwei Objekten zu zeichnen. Dabei handelt es sich entweder um eine Verbindung zwischen einem Attribut innerhalb einer EntityRepresentation und einem Titel einer anderen EntityRepresentation oder eine Verbindung zwischen einem Attribut einer EnittyRepesentation und einer LabelRepresentation. Das QGraphicsTexItem wird genutzt, um eine eventuelle Metric zu definieren (Siehe 3.2.10). Die Connection Klasse ist so programmiert, dass sie bei einer Transformation der verknüpften Elemente automatisch ihre Position anpasst, um deren Transformation zu folgen.

Die LabelRepresentation ist ein einfaches Textwidget (QLabel), welches den Value eines boolean_terms darstellt (Siehe 3.2.10).

Da die EntityRepresentation und die LabelRepresentation beides Widgets sind, müssen diese, wie in 5.2.3 beschrieben, durch ein QGraphicsProxyWidget in die Szene eingefügt werden. Dies wurde in dem Programm durch die Klasse DragBox verwirklicht, welche von QGraphicsProxyWidget erbt. DragBox erweitert den Funktionsumfang der QGraphicsProxyWidget Klasse. Durch die DragBox Klasse ist es möglich, das betroffene Widget per Maus zu verschieben. Somit lassen sich nach Initialisierung der Szene im Anschluss alle Objekte weiterhin verschieben, da die Connection den verbundenen Elementen automatisch folgt.

5.5.2. TEMPLATERULESGRAPHICSVIEW

Wie in Quelltext 9 erkennbar, gibt es oft eine mehrfache Schachtelung von TemplateRules innerhalb einer mvdXML-Datei. Dabei wird dies schnell sehr unübersichtlich, vor allem bei der Verwendung unterschiedlicher Operatoren zwischen einzelnen TemplateRules. Da es sich hier um eine einfache Verschachtelung handelt, wurde sich dazu entschieden diese Verschachtelung auch grafisch zu repräsentieren.

In Abbildung 18 ist die finale grafische Repräsentation des Quelltext 9 dargestellt. Hierbei fällt auf, dass die einzelnen TemplateRules, ähnlich wie die TemplateRule, als ein Rahmen mit TitleBar dargestellt sind. Im Unterschied zur TemplateRule, ist dieser Rahmen jedoch nun eingefärbt und der Titeltext wurde auf den entsprechenden Operator angepasst. Weiterhin ist auch der Hintergrund der Szene im TemplateRulesGraphicsView an die Farbe des Operators angepasst. Folgende Farbzuweisung bestehen:

Tabelle 12 Farbzuweisung Operatoren

Operator	Farbe	RGB Wert Rahmen	RGB Wert Hintergrund	Rahmen	Hintergrund
AND	Grün	(0,110,0)	(217,255,217)		
NAND	Rot	(160,0,0)	(255,217,217)		
OR	Blau	(0,10,160)	(217,220,255)		
NOR	Rot	(160,0,0)	(255,217,217)		
XOR	Violett	(130,0,140)	(255,217,255)		
NXOR	Rot	(160,0,0)	(255,217,217)		
ELSE	Grau	(140, 140, 140)	(235,235,235)		

Genau wie bei der TemplateRule wird auch für die TemplateRules ein eigener View mit zugehöriger Szene erstellt. Da die TemplateRule-Elemente, wie auch die TemplateRules-Elemente, ein View mit entsprechender Szene sind, ist es möglich die entsprechenden Rule- bzw. Rules-Elemente in die Szene der hierarchisch übergeordneten Elemente

einzufügen. Somit entsteht eine übersichtliche und unkomplizierte Übersicht der TemplateRule- und TemplateRules- Elemente, die es ermöglicht, die komplexe Schachtelung gut darzustellen.

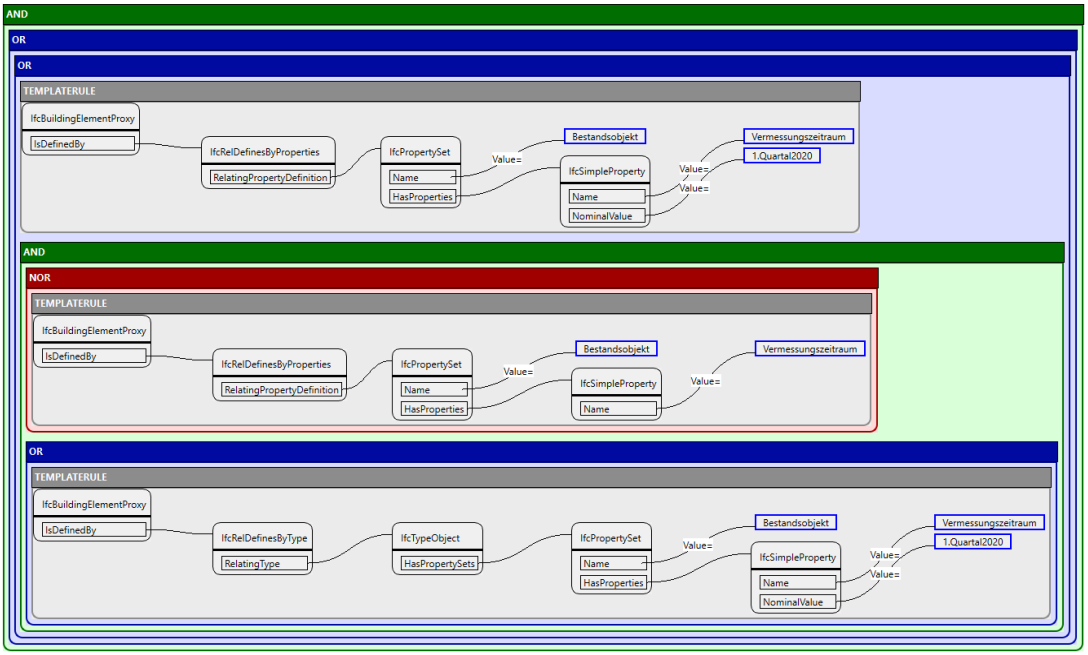

Abbildung 18 Grafische Darstellung von Quelltext 9

Da sich die Klasse TemplateRuleGraphicsView und die Klasse TemplateRulesGraphicsView viele Verhaltensweisen teilen, beispielsweise Rahmen und TitleBar, wurden, wie in Abbildung 25 erkennbar, die gemeinsamen Attribute und Funktionen in der Klasse RuleGraphicsView hinterlegt. TemplateRuleGraphicsView und TemplateRulesGraphicsView erben deshalb von RuleGraphicsView. Somit sind auch verknüpfte Objekte, wie die ResizeEdge- und ResizeBorder-Elemente (Siehe 5.5.3) bei beiden Klassen identisch.

5.5.3. RESIZEEDGE & RESIZEBORDER

Das erstellte Programm soll in Zukunft die Möglichkeit bieten, Elemente einer MVD hinzuzufügen und zu bearbeiten. Deshalb wurden die ResizeEdge- und ResizeBorder-Elemente implementiert, um vorhandene Views in ihrer Geometrie anzupassen und zu verschieben. Die ResizeEdge- und ResizeBorder-Elemente sind als schwarze Rechtecke in Abbildung 19, dargestellt.

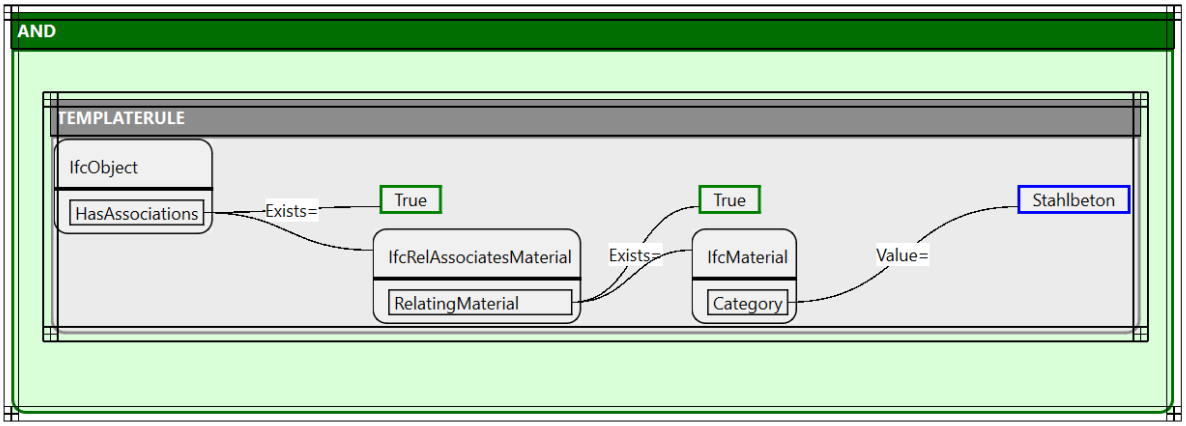

Abbildung 19 Eingeblendete ResizeElements

Dabei folgt das Verhalten der einzelnen Resize-Elemente den grundlegenden Konzepten, die auch von klassischen Windows- oder Mac-Anwendungen bekannt sind. In den Ecken befinden sich die sogenannten ResizeEdge-Elemente. Diese ermöglichen eine Skalierung des Views in horizontaler und vertikaler Richtung. Die ResizeBorder-Elemente sind, abhängig von ihrer Position, stets nur in der Lage den View in einer Achse zu skalieren. Im vollständigen Programm werden die Kanten der Rechtecke ausgeblendet, sodass diese nicht mehr sichtbar sind. Die Möglichkeit der Skalierung wird dann über eine Veränderung des Cursor-Symbols verdeutlicht.

6 FAZIT

Da im Kontext der Infrastruktur noch maßgebliche Standardisierungen fehlen, ist davon auszugehen, dass sich eine geraume Zeit mit Proxy-Elementen und selbstgeschriebenen PropertySets beholfen werden muss. Damit ergibt sich ein inhärenter Zwang, spezialisierte Modellanforderungen und Prüfregeln zu definieren, die spezifisch auf die Wünsche der Infrastrukturplanung zugeschnitten sind. Mit der fortschreitenden Ausweitung der BIM-Methode bei der Deutschen Bahn, wird sich weiterhin auch der Fokus der Modellprüfung ändern. Bisher findet hauptsächlich eine reine Attributprüfung und ein Abgleich derselben mit einem semantischen Objektmodell statt. Zusammenhänge zwischen einzelnen Objekten werden bisher fast ausschließlich händisch geprüft.

Wie auch in anderen Bereichen des Bauwesens, ist der Infrastrukturbereich, insbesondere der Schienenverkehr, stark genormt und reglementiert. Neben DIN und EN existieren innerhalb der Deutschen Bahn weitere Richtlinien (Ril), die bei Gestaltung und Bau eines Streckennetzes beachtet werden müssen. In Zukunft sollte vermehrt daran gearbeitet werden, diese Richtlinien in maschinenlesbare Formate umzuwandeln. Dadurch wäre es möglich, die bisherige händische Prüfung von Bauprozessen weiter zu automatisieren. Hierfür eignet sich das mvdXML-Format ausgezeichnet, da es die vielfältigsten Möglichkeiten bieten, eine Modellanforderung zu spezifizieren.

6.1. ZUSAMMENFASSUNG

Das erstellte Programm ist in der Lage, eine mvdXML-Datei vollständig in eine mvdOWL-Datei umzuwandeln. Die weitergehende Verarbeitung der Ontologie zur Visualisierung ist zwar nicht vollumfänglich durchgeführt wurden, richtet sich allerdings nach den öftest auftretenden Szenarien und kann somit in einem Großteil der Anwendungsfälle genutzt werden. Eine Erweiterung des Programmes wird durch dessen Struktur ermöglicht, sodass in Zukunft die Möglichkeit zur Modifikation von MVDs ergänzt werden kann. Es ist weiterhin zu überlegen, ob eine Umwandlung von mvdOWL zu SHACL und SPARQL Abfragen sinnvoll ist, um ifcOWL-Dateien zu validieren oder zu filtern.

Im Fokus von steigender Benutzung von Ontologien und Semantic Web-Anwendungen im wissenschaftlichen Kontext und in Endanwendungen, stellt die mvdOWL eine sinnvolle Erweiterung des klassischen mvdXML-Formates dar. Durch die Transformation von mvdXML zu mvdOWL gehen keine Daten verloren, die Zusammenhänge werden durch die Nutzung von inversen Eigenschaften jedoch klarer. Somit wird eine einfachere Verarbeitung und ein besseres Verständnis von MVDs ermöglicht und es ist vorstellbar, dass dadurch Anwendungen besser mit MVDs arbeiten könnten.

6.2. AUSBLICK

6.2.1. INFORMATION DELIVERY SPECIFICATION

Wie in 2.1 bereits beschrieben, haben MVDs nicht nur die Aufgabe, Prüfregeln und Modellanforderungen zu spezifizieren, sondern sie werden auch Teil der Export- und Importfunktion von BIM-Programmen. Dadurch entstehen maßgebliche Probleme der Interoperabilität, wenn die Programme für Im- und Export nicht auf die gleichen MVDs zugreifen. Um diese Herausforderung zu lösen, wurde sich seitens BuildingSmart dazu entschieden, in Zukunft MVDs in mehrere Subkomponenten aufzuteilen. Für die Interoperabilität zwischen Programmen sollen in Zukunft drei Konformitätsebenen, welche von BuildingSmart vorgegeben werden, sorgen. Die Modellanforderungen beschreibt in Zukunft die Information Delivery Specification (IDS). Sie ist bisher noch in der Konzeptualisierungsphase, ähnelt aber bisher dem Aufbau einer mvdXML. Es ist deshalb vorstellbar, dass Konzepte, die in dieser Projektarbeit entwickelt wurden, in leicht abgewandelter Form in Zukunft weiterhin auf IDSs angewendet werden können. Außerdem soll die IDS erst mit dem IFC5-Standard eingeführt werden, welcher noch kein Release Datum besitzt.

6.2.2. PROGRAMM ERWEITERUNG

Das erstellte Programm ist bisher so programmiert, dass mvdXML-Dateien nur in Ontologien umgewandelt und dargestellt werden können. Eine neue Erstellung von mvdOWL oder mvdXML-Dateien und deren Modifizierung ist bisher nicht implementiert. Das Programm ist allerdings so geschrieben, dass diese Funktionen in weiteren Arbeitsschritten ergänzt werden können. In Kombination mit der in Kapitel 4.5 erwähnten Möglichkeit der Modellprüfung, wäre es vorstellbar, dass erstellte Programm zu einem vollfunktionsfähigen Modelchecker zu entwickeln.

VII ANLAGENVERZEICHNIS

ANLAGE 1 DIGITALES ANLAGENVERZEICHNIS

- Arbeit (P:)
  - Projektarbeit.docx
  - Projektarbeit.pdf
  - Code
    - Graphics
      - icon.ico
      - icon.png
    - mvd2OWL
      - __init__.py
      - constants.py
      - core.py
      - visualization.py
    - tests
      - __init__.py
      - test.py
    - main.py
  - mvdXML_Beispiele
    - bim-q-test.mvdXML
    - RelAssociatesMaterial.mvdxml
    - IFC4precast_V1.01.mvdxml
  - mvd2OWL_precompiled.zip

ANLAGE 2 WEITERE ABBILDUNGEN

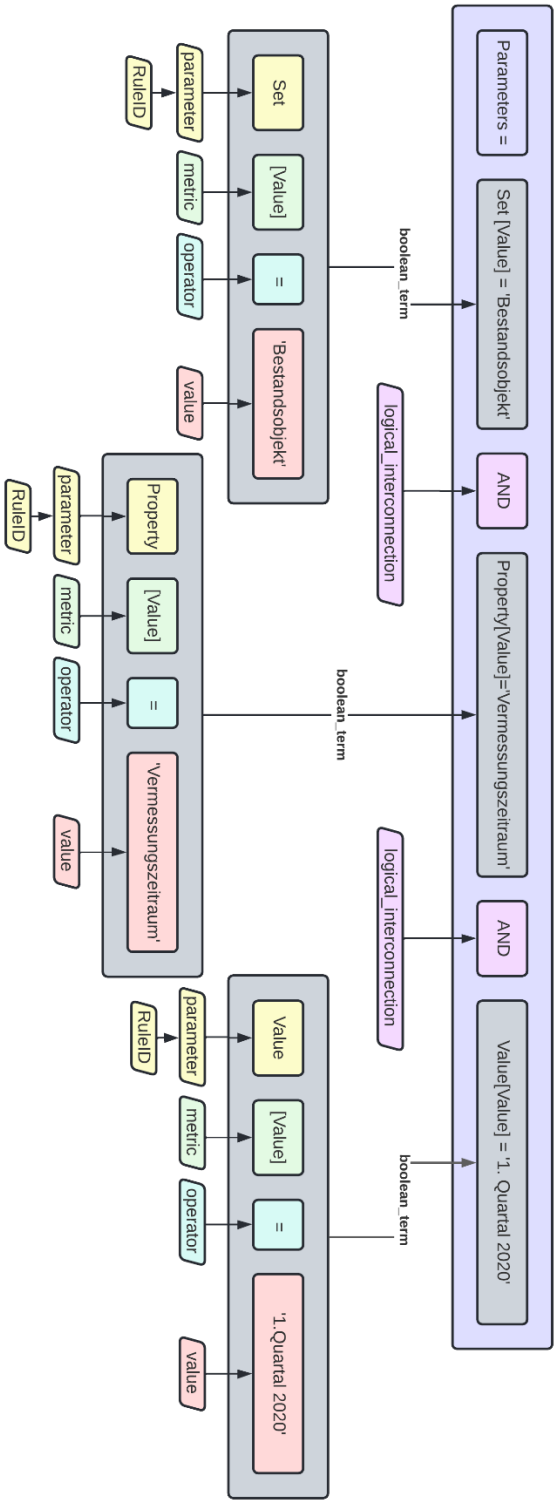

Abbildung 20 Zerlegung Parameter

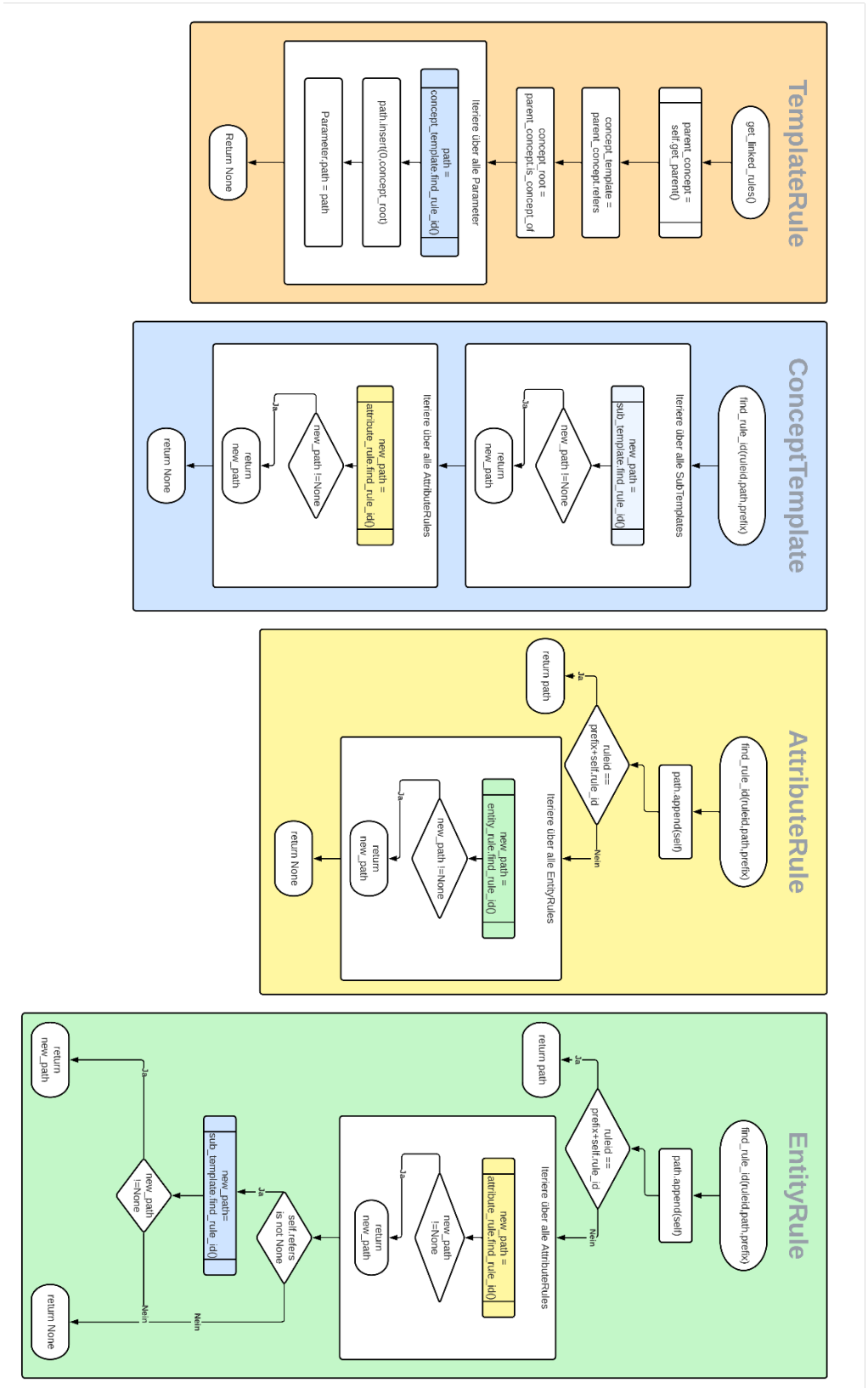

Abbildung 21 Flussdiagramm get_linked_rules()

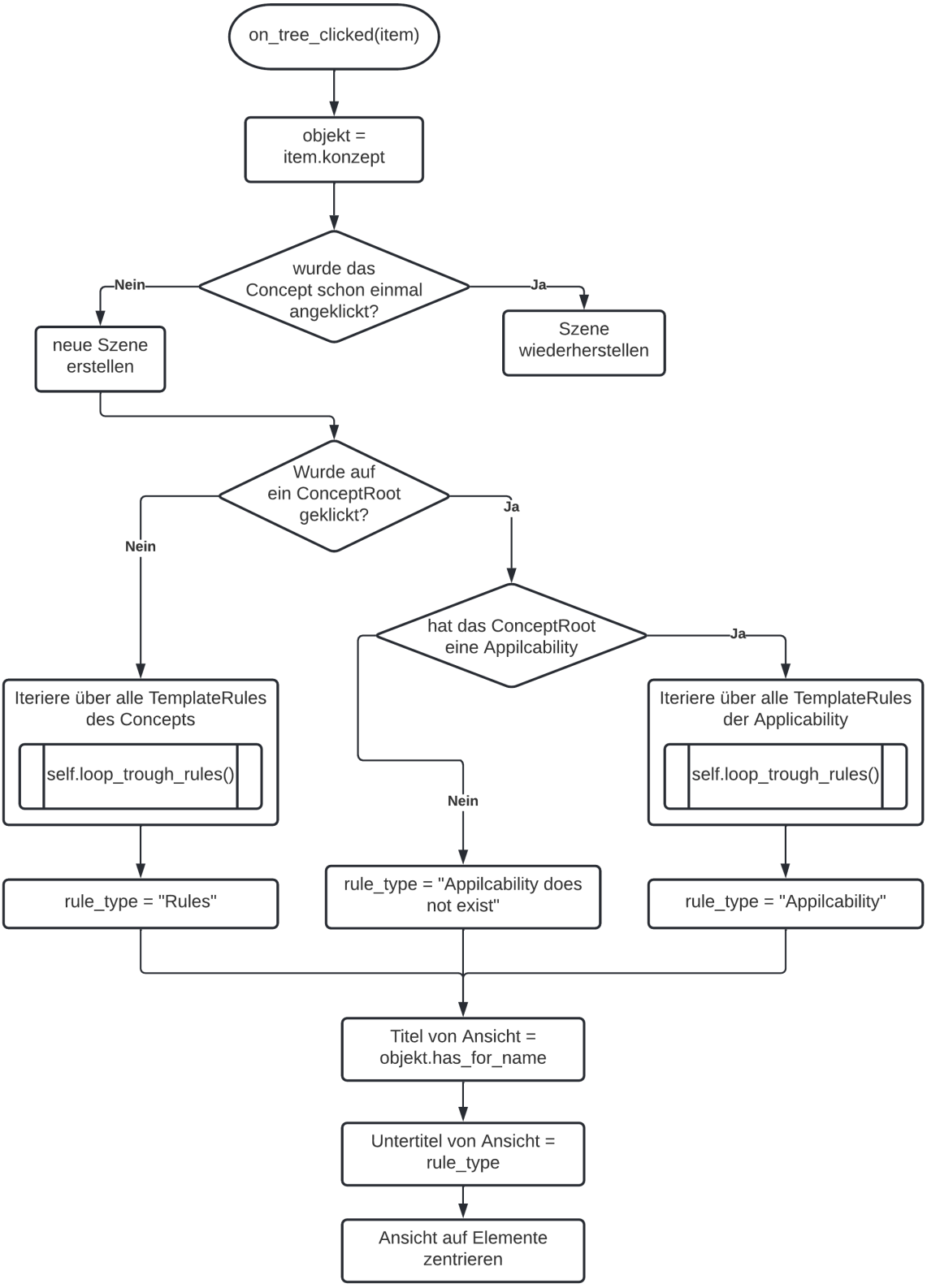

Abbildung 22 Flussdiagramm on_tree_clicked()

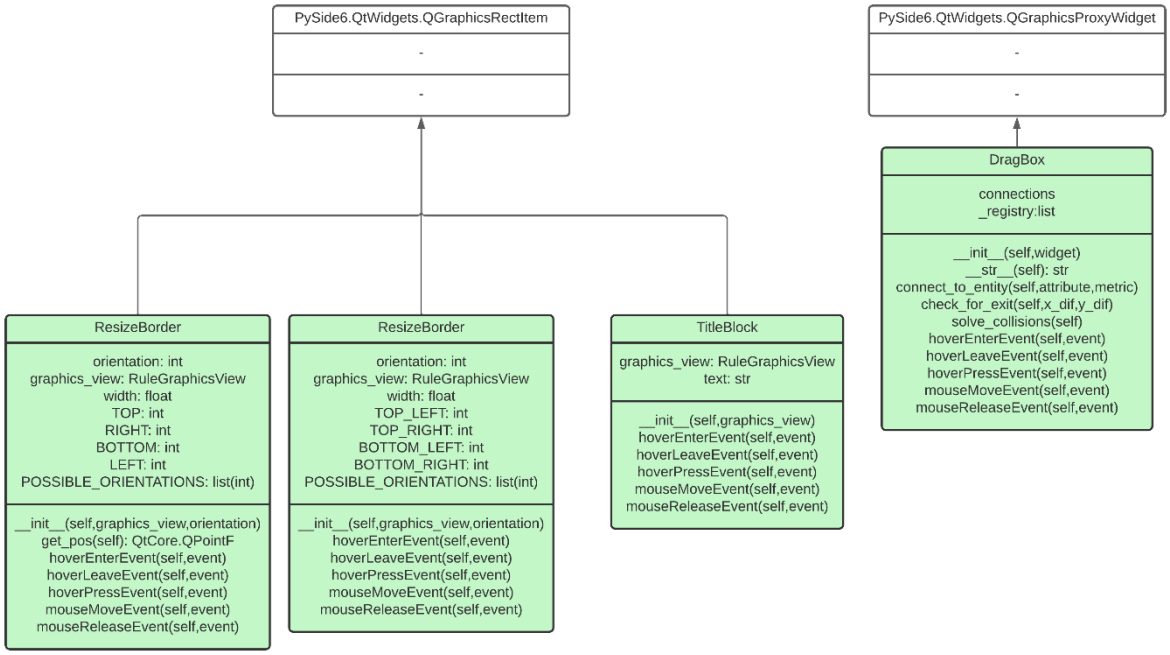

Abbildung 23 UML QGraphicsItems

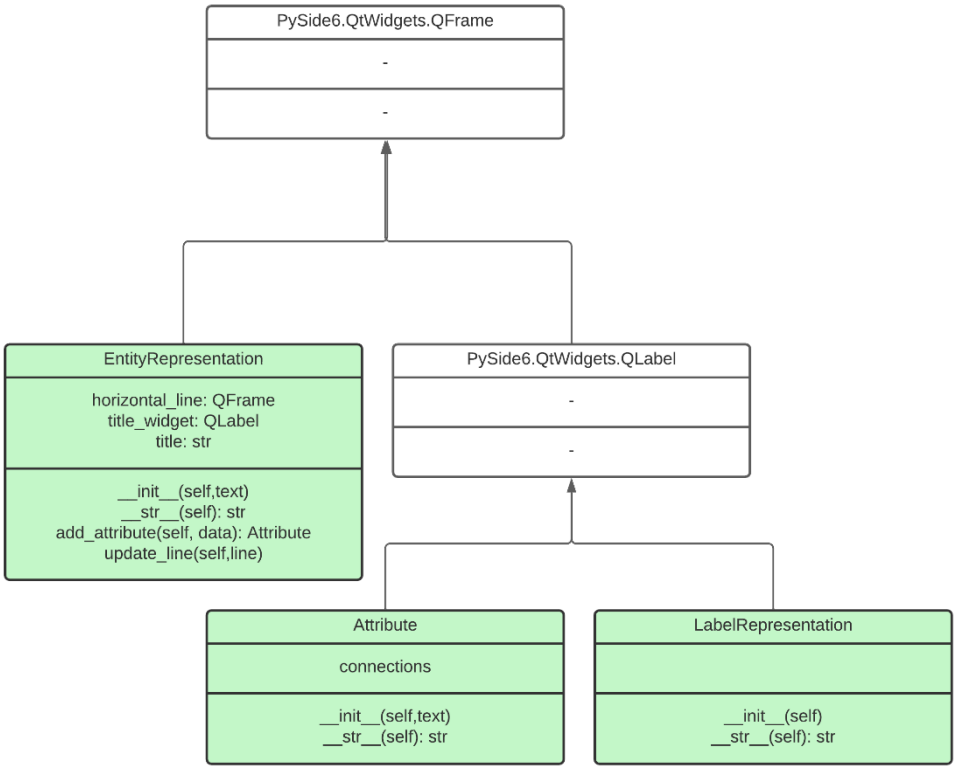

Abbildung 24 UML TemplateRule Elemente

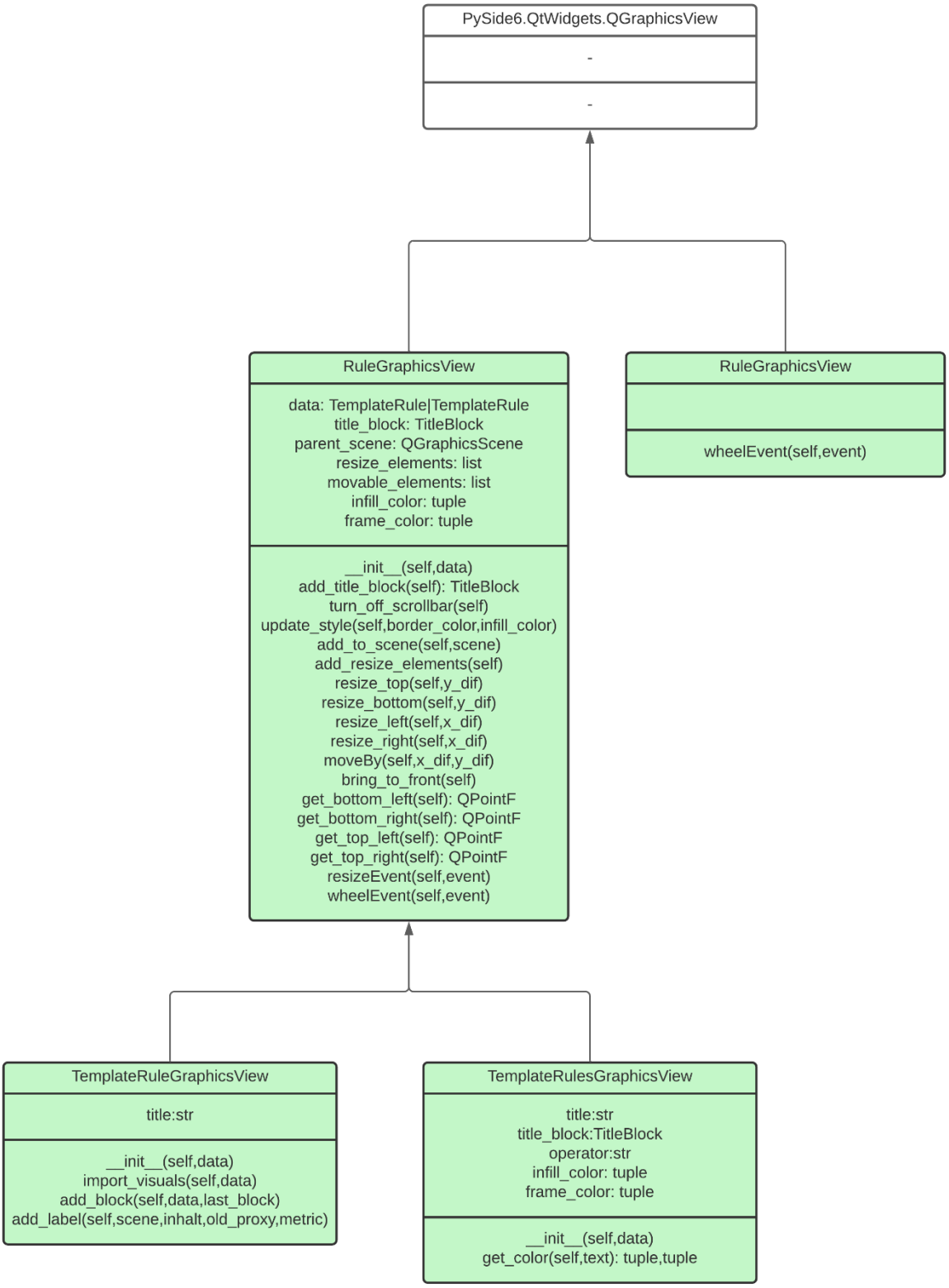

Abbildung 25 UML QGraphicsView

7 LITERATURVERZEICHNIS

Baumgärtel, K. & Pirnbaum, S. (2016). Automatische Prüfung und Filterung in BIM mit Model View Definitions. In T. Berthold, S. Brand & C. Schiermeyer (Vorsitz), *Institutionelles Repositorium der Leibniz Universität Hannover.* Symposium im Rahmen der Tagung von Leibniz Universität Hannover, Hannover.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. & Yergeau, F. (2013, 7. Februar). Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C. https://www.w3.org/TR/xml/*, last accessed 14.03.2022.

buildingSMART International Ltd. (2022). Model View Definition (MVD). https://technical.buildingsmart.org/standards/ifc/mvd/*, last accessed 15.03.2022.

Chipman, T., Liebich, T [Thomas] & Weise, M. (2012). mvdXML Version 1.0 Final. buildingSMART International Ltd. https://standards.buildingsmart.org/MVD/RELEASE/mvdXML/v1-0/mvdXML_V1-0.pdf*, last accessed 14.03.2022.

Chipman, T., Liebich, T [Thomas] & Weise, M. (2016). mvdXML Version 1.1 Final. buildingSMART International Ltd. https://standards.buildingsmart.org/MVD/RELEASE/mvdXML/v1-1/mvdXML_V1-1-Final.pdf*, last accessed 18.01.2022.

Destatis. (2022, 25. Februar). VGA Monitor Deutschland-Beschäftigung. https://service.destatis.de/DE/vgr-monitor-deutschland/beschaeftigung.html*, last accessed 14.03.2022.

ISO (29. Januar 2016). *Industrial automation systems and integration: Product data representation and exchange — Part 21: Implementation methods: Clear text encoding of the exchange structure* (ISO 10303-21:2016). https://www.iso.org/standard/63141.html

ISO (04.2020). *Information container for linked document delivery: Exchange specification - Part 1: Container* (ISO 21597-1:2020). https://www.iso.org/standard/74389.html

Jean-Baptiste, L. (2021). *Ontologies with Python: Programming OWL 2.0 Ontologies with Python and Owlready2*. *Springer eBook Collection*. Apress. https://doi.org/10.1007/978-1-4842-6552-9

Knublauch, H. & Kontokostas, D. (2017, 20. Juli). Shapes Constraint Language (SHACL). https://www.w3.org/TR/shacl/*, last accessed 15.03.2022.

Luttun, Johan and Krijnen, Thomas (2021). An Approach for Data Extraction, Validation and Correction Using Geometrical Algorithms and Model View Definitions on Building Models. In Toledo Santos, Eduardo and Scheer, Sergio (Hrsg.), *Proceedings of the 18th International Conference on Computing in Civil and Building Engineering* (S. 529–543). Springer International Publishing.

Pauwels, P. & Terkaj, W. (2016). EXPRESS to OWL for construction industry: Towards a recommendable and usable ifcOWL ontology. *Automation in Construction*, *63*, 100–133. https://doi.org/10.1016/j.autcon.2015.12.003

Richter, S. (2012, 25. Dezember). lxml Website, last accessed 19.03.2022.

Srinath, K. R. (2017). Python – the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, *4*(12), 354–357.

W3C. (2022, 14. März). HTML. https://html.spec.whatwg.org/multipage/*, last accessed 15.03.2022.

W3C OWL Working Group (Hrsg.). (2012, 11. Dezember). *OWL 2 Web Ontology Language Document Overview (Second Edition)*. https://www.w3.org/TR/owl2-overview/

W3C SPARQL Working Group. (2013, 21. März). SPARQL 1.1 Overview. https://www.w3.org/TR/sparql11-overview/*, last accessed 15.03.2022.

Weise, M., Liebich, T [T.], Nisbet, N. & Benghi, C. (2016). IFC model checking based on mvdXML 1.1 Weise M., Liebich T., Nisbet N. In Z. Turg & R. Scherer (Vorsitz), *ECPPM,* Limassol, Cyprus.