Gottfried Wilhelm Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik

Unterstützung der Manuskripterstellung mithilfe strukturiertem wissenschaftlichem Wissen

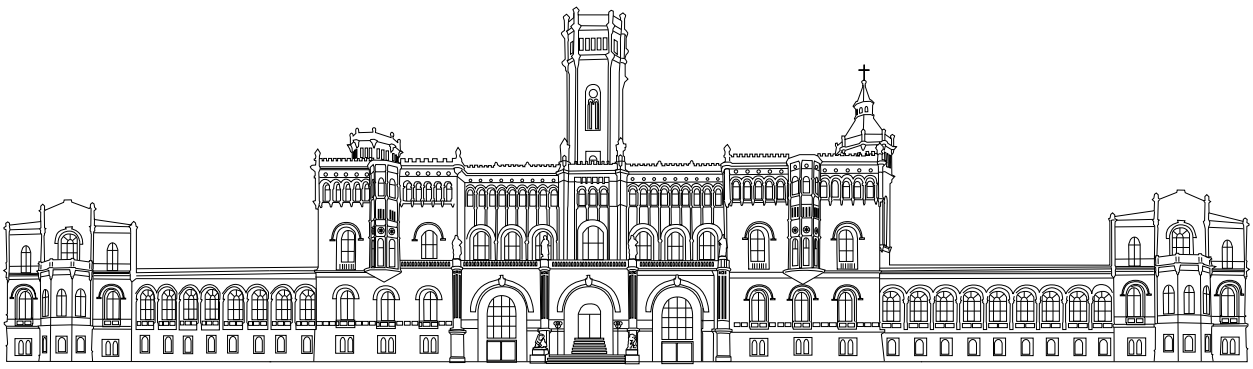
Eine Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science in Informatik

von

Oliver Ludwig
Matrikelnummer: 10032782
E-Mail: o.ludwig@stud.uni-hannover.de

Erstprüfer: Prof. Dr. Sören Auer
Zweitprüfer: Dr. Markus Stocker
Betreuer: Dr. Markus Stocker

15. März 2024



Erklärung der Selbständigkeit

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind, und die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen hat.

Oliver Ludwig

Hannover, den 15.03.2024

Erklärung zur Plagiatsprüfung

Mit der Übermittlung meiner Arbeit auch an externe Dienste zur Plagiatsprüfung durch Plagiatssoftware erkläre ich mich einverstanden.

Oliver Ludwig

Hannover, den 15.03.2024

Danksagung

Zunächst möchte ich meinen tiefsten Dank an meinen Betreuer, Dr. Markus Stocker, aussprechen, der sich stets die Zeit genommen hat, um meine zahlreichen Fragen so umfassend wie möglich zu beantworten. Des Weiteren möchte ich mich bei Prof. Dr. Sören Auer bedanken, der diese Arbeit ermöglicht hat. Den restlichen Teil möchte ich meiner Familie widmen, die mich durch alle Lebensphasen begleitet hat. Insbesondere möchte ich dabei meinen Eltern für ihre kontinuierliche Unterstützung in meiner Ausbildung und meinen Interessen danken.

Abstract

In recent years, the number of published scientific publications has significantly increased. As most publications are only published as an written Manuscript, collected results are challenging for machines to utilize. Additionally comparisons of different works are time-consuming and require human intervention. To address these challenges, the Open Research Knowledge Graph (ORKG) was developed to structure informations from scientific publications into a machine-readable knowledge graph. This Thesis explores how authors can be supported by software that generates appropriate visualizations when structured data is given. The aim is to develop an implementation that encourages authors to integrate results of their scientific work into knowledge graphs like the ORKG. A Microsoft Word Add-In created as a part of this work, demonstrates how visualizations can be automatically generated using structured data and ORKG templates. It was observed that such an implementation can greatly support users and efficiently generate representations. A notable feature of the generated visualizations is their ability to easily be edited within the document without requiring regeneration. Despite the successful results, further adjustments to the implemented system are necessary to ensure full support of all functionalities and a sufficient amount of visualizations.
Keywords: ORKG, structured data, scientific knowledge, visualizations, manuscript writing

Zusammenfassung

In den letzten Jahren ist die Anzahl veröffentlichter wissenschaftlicher Publikationen stark angestiegen. Da die meisten Veröffentlichungen in schriftlicher Form vorliegen, können gesammelte Ergebnisse nur schwer von Maschinen verwendet werden. Auch der Vergleich verschiedener Arbeiten ist zeitintensiv und erfordert ein menschliches Eingreifen. Um diesen Herausforderungen zu begegnen, wurde der Open Research Knowledge Graph (ORKG) entwickelt, um Wissen aus wissenschaftlichen Publikationen in einen maschinenlesbaren Wissensgraphen zu strukturieren. Diese Arbeit untersucht, wie Autoren mithilfe einer Software unterstützt werden können, die passende Visualisierungen generiert, wenn strukturierte Daten eingegeben werden. Ziel ist es, eine Implementierung zu entwickeln, die Autoren dazu anregt, Ergebnisse ihrer wissenschaftlichen Arbeiten in Wissensgraphen wie dem ORKG zu integrieren. Anhand eines im Rahmen dieser Arbeit erstellten Microsoft Word Add-Ins wird gezeigt, wie mithilfe strukturierter Daten und ORKG-Templates Visualisierungen automatisch erstellt werden können. Dabei wurde beobachtet, dass eine solche Umsetzung eine große Unterstützung für den Nutzer sein kann und effiziente Darstellungen generiert werden können. Eine besondere Eigenschaft der generierten Visualisierungen ist, dass sie sich leicht innerhalb des Dokuments verändern lassen, ohne diese neu generieren zu müssen. Trotz der erfolgreichen Ergebnisse sind weitere Anpassungen des implementierten Systems erforderlich, um eine vollständige Unterstützung aller Funktionalitäten und eine ausreichende Menge an Visualisierungen zu gewährleisten.
Stichworte: ORKG, strukturierte Daten, wissenschaftliches Wissen, Visualisierungen, Manuskripterstellung

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel der Arbeit . . . 2
- 1.3 Struktur der Arbeit . . . 2

2 Hintergrund **4**
- 2.1 ORKG . . . 4
  - 2.1.1 ORKG-Templates . . . 5
- 2.2 Microsoft Word . . . 6
- 2.3 Wichtige Dateiformate . . . 7
  - 2.3.1 XML . . . 7
  - 2.3.2 Office Open XML . . . 8
  - 2.3.3 JSON . . . 10
  - 2.3.4 YAML . . . 11
- 2.4 Verwandte Arbeiten . . . 12
  - 2.4.1 SciKGTeX . . . 12
  - 2.4.2 ORKG Python- und R-Bibliothek . . . 13
  - 2.4.3 Datenerfassung nach der Veröffentlichung . . . 13

3 Ansatz **15**
- 3.1 Einordnung innerhalb des ORKG Systems . . . 15
- 3.2 Aufgabe und Ziele des Systems . . . 17
- 3.3 Systemumsetzung . . . 18
  - 3.3.1 Interaktion mit der Schreibumgebung . . . 18
  - 3.3.2 Wahl der Textverarbeitungssoftware . . . 19
- 3.4 Technische Umsetzung . . . 20
- 3.5 Evaluation . . . 24
- 3.6 Use Cases . . . 24

3.7 Testausführung 25
3.8 Testauswertung 26

4 Diskussion **29**
4.1 Probleme bei der Implementierung 29
4.1.1 Versionsunterschiede und Inkompatibilität 29
4.1.2 Konventionen bei der Erstellung eines Templates 31
4.1.3 Aufwendige Erstellung von Templates 32
4.2 Stärken 32
4.2.1 Weite Versionskompatibilität 33
4.2.2 Wiederverwendbarer Code 33
4.2.3 Einfache Benutzung und Anpassungsfähigkeit 35
4.3 Schwächen 36
4.3.1 Schreiben von Spezifikationen 36
4.3.2 Konventionen und fehlende Daten 37
4.3.3 Templates und Funktionserweiterungen 38
4.4 Aufgriff der Forschungsfrage 38

5 Fazit und Ausblick **40**
5.1 Fazit 40
5.2 Ausblick 41

Bibliography **43**

Abbildungsverzeichnis

2.1 Beispiel einer Publikation in einen Wissensgraphen. Visualisierung mithilfe von https://rdfshape.weso.es/dataInfo. 5
2.2 Ausschnitt des „Student's t-test“ Templates auf der ORKG Website. 5
2.3 Grafik aus Boyer et al. (2002) , welche ein Beispiel für XML-Knoten Syntax angibt (links) und die dazugehörige Hierarchie als Baum-Datenstruktur (rechts). 8

3.1 Ablauf zur Nutzung des zu implementierenden Programms mit den ORKG Funktionalitäten. 16
3.2 Möglichkeiten zur Umsetzung der Interaktion von Anwendung mit der Schreibumgebung. Dargestellt sind die Interaktion einer entkoppelte Software (links) und einer Erweiterung (rechts). 19
3.3 Kommentar auf die Versionsüberprüfung von Word bei Benutzung von Fluent UI React. 21
3.4 Geöffnetes ORKG Add-In Fenster in Word. 25
3.5 Vergleich einer generierten Kastengrafik (A) mit einer Referenz-Visualisierung (B) des „Student's t-test“ Use Case aus Haddad et al. (2016) in Microsoft Word. 26
3.6 Kastengrafik Visualisierung des „Student's t-test“ Use Case des Iris Datensatzes in Microsoft Word. 27
3.7 Vergleich einer generierten Tabelle (A) und der Referenztabelle (B) aus Thießen et al. (2023) 28

4.1 Fluent UI React Fehlermeldung. 30
4.2 Kommentar auf die Versionsüberprüfung von Word bei Benutzung von Fluent UI React. 30
4.3 Vergleich von einer Funktion mit API-Aufruf zu Tabellen Generierung (links) und erste paar Zeilen des verwendeten Tabellen-Template (rechts) bezüglich ihrer Länge. 33
4.4 Möglichkeit zum Vereinfachen von YAML-Spezifikationen. 37

Tabellenverzeichnis

4.1 Generierte Latex Tabelle aus dem Table-Template. (Nur dieser Untertitel wurde hinzugefügt). 36

Verzeichnis für Codeauflistungen

2.1 Ausschnitt einer OOXML-Datei um in Microsoft Word einen einfachen Text anzuzeigen. 9
2.2 Ausschnitt einer beispielhaften JSON-Datei. 10
2.3 Ausschnitt einer beispielhaften JSON-LD Datei. 11
2.4 Ausschnitt einer beispielhaften YAML-Datei. 12
2.5 Python Code für ORKG-Template Materialisierung und Verwendung. . . . 14
3.1 YAML Spezifikation für ein beispielhaftes ORKG-Template. 23
3.2 Ausschnitt einer beispielhaften JSON-LD Datei für die Implementierung. . 23
4.1 JSON Leaderboard Beispiel Ausschnitt Fortsetzung. 32
4.2 Beispiel für ein Table-Template für Latex. 35
4.3 Typescript Code für die Funktion insertRowLatex. 35

Akronyme

DOI Digital Object Identifier

ECMA European Computer Manufacturers Association

HTML Hypertext Markup Language

IRI Internationalized Resource Identifier

JSON JavaScript Object Notation

JSON-LD JavaScript Object Notation for Linked Data

LLM Large Language Model

OOXML Office Open XML

ORKG Open Research Knowledge Graph

RDF Resource Description Framework

VBA Visual Basic for Application

XML Extensible Markup Language

YAML YAML Ain't Markup Language

Kapitel 1

Einleitung

1.1 Motivation

Im Hinblick auf die stetig wachsende Anzahl an veröffentlichten wissenschaftlichen Beiträgen, wurde an vielen Möglichkeiten gearbeitet um mit einem solchen Wachstum umzugehen. Der traditionelle Ansatz, Informationen ausschließlich in schriftlicher Form zu veröffentlichen ist nicht mehr rentabel aufgrund des enormen Wachstum an Publikationen. Unter diesen Umständen wuchsen die Bestreben für alternativen Vorgehensweisen. Viele dieser Ideen ließen sich dabei von den sogenannten FAIR Prinzipien inspirieren, nach welchen die Daten auffindbar, zugänglich, interoperabel und wiederverwendbar gespeichert werden sollen (Wilkinson et al., 2016). Unter Berücksichtigung dieser Prinzipien entstand so beispielsweise die Plattform „Papers With Code'', welche eine offene Gemeinschaft zur Bereitstellung von wissenschaftlichen Arbeiten im Thema Maschinelles Lernen ist. Zusätzlich werden auf dieser Plattform zu einem schriftlichen Dokument Informationen wie der verwendete Code oder verwendete Datensätze hinterlegt[1], um die Wiederverwendbarkeit dieser zu gewährleisten. In einem weiteren Beispiel wurde untersucht, wie man wissenschaftliche Erkenntnisse mithilfe von Wissensgraphen darstellen kann, um diese von Maschinen lesbar zu machen (Fathalla et al., 2017). Daraufhin wurde mithilfe diesem Prinzip auch der Open Research Knowledge Graph (ORKG) entwickelt, welcher eine solche Implementierung eines Wissensgraphen ist. Um das Wachstum des ORKG zu unterstützen wurde ähnlich wie bei „Papers With Code'' ein Open Source Konzept überlegt, bei dem jederzeit mit einem kostenlosen Account Beiträge hinzugefügt werden können (Auer et al., 2020). Um für Autoren weitere Anreize zu setzen, ihre wissenschaftlichen Kenntnisse im ORKG zu veröffentlichen, wurden eine

[1] https://paperswithcode.com/about (Abgerufen: 10.03.2024)

Reihe von unterstützenden Systemen implementiert. Während alle diese Systeme direkt an der Generierung von strukturierten Daten beteiligt sind, fehlt es dennoch an Systemen, welche Daten aus dem ORKG zum Vorteil der Nutzer verwenden. Bei der Wahl eines solchen Systems ist es dabei besonders wichtig, dass es die Autoren direkt unterstützt. Aufgrund der Wichtigkeit von Abbildungen, insbesondere der Visualisierungen von Ergebnissen, ist ein System zur Erstellung von Visualisierungen unter der Benutzung von generierten strukturierten Daten ein geeignetes System. Durch eine einfache und effektive Umsetzung könnte man den den Nutzern so viel Zeit bei der Erstellung von Abbildungen abnehmen. Zusätzlich würde ein solches System bei der Nutzung von ORKG-Daten indirekt für weitere Beiträge am ORKG beteiligt sein.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es zu untersuchen, inwiefern es möglich ist, mithilfe einer Software und strukturierten wissenschaftlichen Wissens, Autoren bei dem Verfassen von wissenschaftlicher Arbeiten zu unterstützen. Dazu werden die Anforderungen eines solchen Systems beschrieben und anschließend eine Implementierung in Form eines Word Add-Ins genannt. Durch das Testen dieser Anwendung und das Vergleichen mit Referenzresultaten, soll ermittelt werden wie präzise und hilfreich generierte Visualisierungen sind. Anhand der Resultate und der gesammelten Erfahrungen bei der Entwicklung des Systems kann dann ein Fazit über die Effektivität einer solchen Anwendung bei der Unterstützung in der Manuskripterstellung, getroffen werden. Die folgende Forschungsfrage soll in dieser Arbeit beantwortet werden, zusätzlich werden weitere Unterfragen angeführt, welche während der Arbeit ebenfalls beantwortet werden:

- Wie kann ein solches System einen Autor beim Schreibprozess unterstützen?
  - Wie können ORKG-Daten mithilfe einer Software visualisiert werden?
  - Wie sieht eine Beispielsimplementierung aus (ORKG Word Add-In)?
  - Welche Vor- und Nachteile lassen sich aus der Implementierung schließen?

1.3 Struktur der Arbeit

Dieser Abschnitt beschreibt den Aufbau der Arbeit, welcher aus fünf Abschnitten besteht. In Abschnitt 2 werden wichtige Technologien und verwandte Arbeiten genannt,

welche für ein besseres Verständnis dieser Arbeit benötigt werden. Anschließend wird in Abschnitt 3 das zu implementierende System im Detail beschrieben und eine passende Umsetzung genannt. Diese Umsetzung wird dabei mit einem Testdurchlauf ausgewertet. Der Abschnitt 4 behandelt die Diskussion, in welcher die Implementierung bezüglich ihrer Probleme und ihrer Eigenschaften bewertet wird. Zudem wird die Forschungsfrage erneut aufgegriffen und beantwortet. Zuletzt wird im Abschnitt 5.1 ein Fazit gezogen und weiterführende Ansätze und mögliche Arbeiten genannt.

Kapitel 2

Hintergrund

In diesem Abschnitt werden zentrale Technologien vorgestellt, welche im Rahmen dieser Arbeit zum Einsatz kamen. In Abschnitt 2.1 wird zuallererst der Open Research Knowledge Graph (ORKG) erläutert. Dabei wird insbesondere auf die Erstellung und Verwendung von ORKG-Templates eingegangen. Anschließend folgt in Abschnitt 2.2 ein kurzer Einblick in die Geschichte der Software Microsoft Word und dessen JavaScript Schnittstelle. Danach werden in Abschnitt 2.3 die wichtigsten Dateiformate, welche in dieser Arbeit verwendet wurden, beschrieben. Als letztes werden im Abschnitt 2.4 verwandte Arbeiten genannt auf welche diese Arbeit aufbaut.

2.1 ORKG

Der ORKG ist ein Wissensgraph für wissenschaftliche Publikationen (Auer et al., 2020). Ein Wissensgraph ist eine Graphstruktur, in der Informationen in Form von Knoten und Kanten gespeichert werden. Beispielsweise kann in solchen Graphen für eine Publikation ein Problem oder ein Vorgehensweise beschrieben werden. Die Publikation wird nun als Knoten dargestellt und besitzt zwei benannte Kanten „hasApproach“ und „adresses“, die auf ein Vorgehen beziehungsweise ein Problem zeigen. Die Abbildung 2.1 stellt einen solchen Graphen dar. Im Vergleich zu rein traditionellen Veröffentlichungen in Form von einzelnen schriftlichen Dokumenten gewährleistet eine solche Beschreibung von Wissen, dass Informationen sowohl für Menschen als auch für Maschinen lesbar sind. Des Weiteren wird auch die Suche nach wissenschaftlichen Arbeiten sowie das Vergleichen von Resultaten stark erleichtert (Auer et al., 2020). Das Sammeln von Beiträgen für den ORKG basiert auf einen Crowdsourcing Ansatz was bedeutet, dass jeder Nutzer Beiträge erstellen kann. Das Ausfüllen von

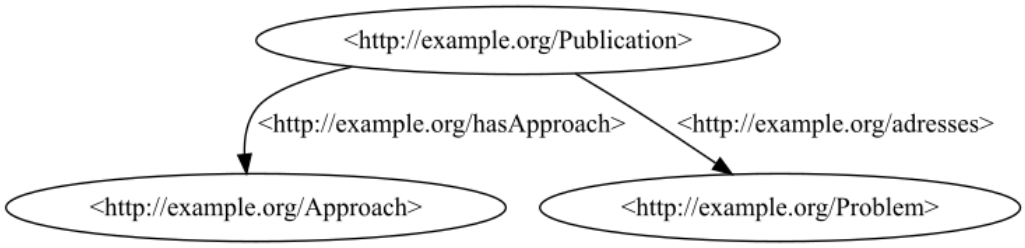


Abbildung 2.1: Beispiel einer Publikation in einen Wissensgraphen. Visualisierung mithilfe von https://rdfshape.weso.es/dataInfo.

Daten ist über einen Account auf der offiziellen Website[1] möglich. Außerdem ist es möglich Templates zu erstellen um ähnliche Beiträge schneller ausfüllen zu können.

2.1.1 ORKG-Templates

ORKG-Templates sind eine wichtige Funktion, um ein bestimmtes Datenlayout für eine Gruppe von Beiträgen festzulegen. Als Beispiel kann das „Student's t-test" Template[2], welches in Abbildung 2.2 zu sehen ist, dienen. Das Template legt fest, dass

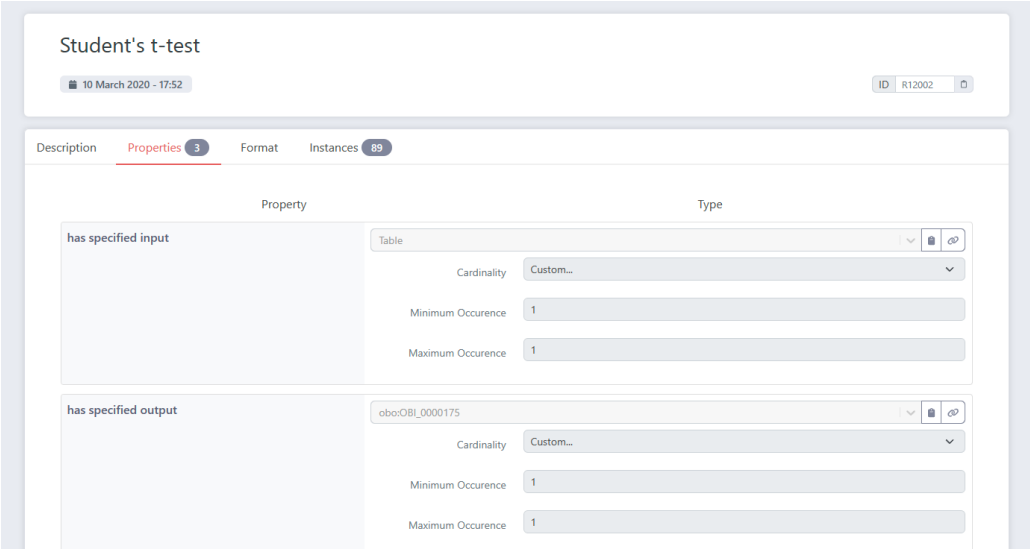


Abbildung 2.2: Ausschnitt des „Student's t-test" Templates auf der ORKG Website.

[1] https://orkg.org/ (Abgerufen: 10.03.2024)
[2] https://orkg.org/template/R12002 (Abgerufen: 10.03.2024)

ein Objekt vom Typ „Student's t-test“ die Attribute „has specified input“, „has specified output“ und „has dependent variable“ besitzen muss. Des Weiteren werden die Kardinalitäten und der Typ von Werten für jedes Attribut festgelegt. So kann man zum Beispiel in der Abbildung 2.2 erkennen, dass das Attribut „has specified output“ genau eine Tabelle als Wert haben muss. Zudem erhält jedes Template eine Identifikation (ID), welche in der Abbildung oben rechts zu erkennen ist.

2.2 Microsoft Word

Die Software Microsoft Word erschien 1983 für das Betriebssystem MS-DOS (Tsang, 2000) und wurde später zusätzlich für MacOS, Android und iOS veröffentlicht. Zu damaliger Zeit unterschieden sich anderer Textverarbeitungssoftwares vor allem dadurch, dass es bei diesen erst nach der Bearbeitung möglich war die vom Benutzer erstellte Formatierung anzuzeigen. Word hingegen setzte auf eine Grafische Benutzeroberfläche (GUI), welche während der Bearbeitung bereits den auszugebenden Text anzeigt. Durch weitere Neuerungen und insbesondere der Unterstützung von Laserdruckern gewann Microsoft Word stark an Popularität und wurde so ab Anfang der 90er zum Marktführer (Peterson, 2001). Auch heute noch ist Word die am meisten verbreitete Textverarbeitungssoftware und sei nach Angaben einer Studie von 2010 seiner Konkurrenz bislang noch stark voraus [3].
Im Laufe der Geschichte von Word wurden eine Vielzahl an weiteren Funktionen veröffentlicht, wie zum Beispiel das Erstellen von Tabellen, Diagrammen, automatischen Nummerierungen und vieles mehr. Besonders wichtig für die folgenden Abschnitte dieser Arbeit ist dennoch die Veröffentlichung einer JavaScript Schnittstelle. Während man zuvor komplexere Funktionen in der Regel nur mit der integrierten Skriptsprache Visual Basic for Application (VBA) realisieren konnte, wurde ab der Version „Word 2013“ auch eine Alternative angeboten, welche es einem Nutzer erlaubt durch eine externe Anwendung mithilfe mehrerer JavaScript APIs Inhalte in Word Dokumenten einzufügen oder zu verändern. Der Umfang dieser JavaScript Schnittstelle ist im Vergleich zu dem der VBA-Funktionen zwar stark begrenzt, dennoch bietet eine Verwendung von JavaScript gegenüber VBA andere Vorteile. Einer dieser Vorteile ist die Plattformunabhängigkeit. So wird VBA anders als JavaScript beispielsweise nicht in der Web- oder iOS Version von Word unterstützt. Des Weiteren lässt sich mithilfe von JavaScript, ihrer Schnittstelle, HTML und CSS auch eine Web-App erstellen, welche unabhängig von Word entwickelt werden kann

[3] https://www.webmasterpro.de/news/international-openoffice-market-shares/ (Abgerufen: 10.03.2024)

und somit leichter auf eine Vielzahl von Nutzern verteilt werden kann. Mit weiteren Veröffentlichungen von Word wie Word 2016, 2019 und 2021 nahm die Menge an unterstützten JavaScript Funktionen der Schnittstelle stark zu. Um die Verfügbarkeit einzelner Funktionen übersichtlicher zu halten wurden einzelne Funktionen zu einem API-Set zusammengefügt. Abhängig der Word-Version wird nur eine gewisse Anzahl an API Sets unterstützt. Die Menge der unterstützten API Sets neuerer Versionen bildet dabei derzeit immer eine Obermenge der unterstützten Sets älterer Versionen[4]. Aus funktionaler Sicht lassen sich eine Vielzahl an neueren Funktionen dennoch auch mithilfe älterer Funktionen umsetzen, da es mit „Word 2013“ bereits möglich war Word Objekte in Form von OOXML-Dateien in ein bestehendes Dokument zu laden.

2.3 Wichtige Dateiformate

Besonders wichtig während der Implementierung waren die Datenformate: Extensible Markup Language (XML), JavaScript Object Notation (JSON) und YAML Ain't Markup Language (YAML). Im Weiteren werden diese jeweils kurz beschrieben. Dabei wird besonders auf die spezifischen Syntaxen eingegangen.

2.3.1 XML

XML ist eine Auszeichnungssprache, welche auf der Struktur der Standard Generalized Markup Language (SGML) aufgebaut ist. XML ermöglicht eine hierarchische Abspeicherung von Daten in Form einer Textdatei. Es ist sowohl menschen- als auch maschinenlesbar.
Ein XML-Dokument wird immer mit einer XML-Deklaration eingeleitet, in welcher die verwendete Version angegeben wird. Optional folgt daraufhin eine Dokumentdeklaration „<!DOCTYPE ...>“, die die Grammatik angibt. Beispielsweise gibt „<!DOCTYPE html>“ an, dass es sich bei dem Dokument um HTML-Spezifische XML-Syntax handelt[5]. Demnach wird die Menge der möglichen Elemente beschränkt und Regeln für die bestimmten Element-Hierarchien definiert (Bray et al., 2008). Anschließend folgt der Wurzelknoten (root node), mit welchem der Rest des Inhalts eingeleitet wird. Ein Knoten X wird durch einen Starttag „<X>“ und ein Endtag „</X>“ dargestellt. Zwischen dem Starttag und dem Endtag können sich Text oder weitere Knoten (Kind Elemente) befinden. Alternativ kann ein Knoten ohne Kind

[4]https://learn.microsoft.com/en-us/javascript/api/requirement-sets?view=common-js-preview (Abgerufen: 10.03.2024)

[5]https://html.spec.whatwg.org/multipage/syntax.html#the-doctype (Abgerufen: 10.03.2024)

Elemente oder Text auch mit einem leeren Tag „<X/>“ dargestellt werden (Boyer, 2001). Wie man in Abbildung 2.3 sehen kann, lässt sich jede XML-Struktur aufgrund ihres hierarchischen Aufbaus als Baum darstellen.

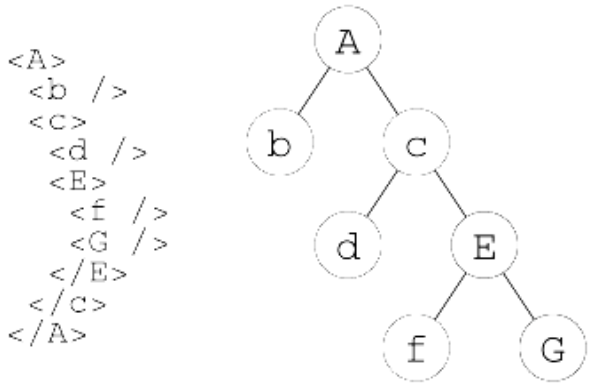


Abbildung 2.3: Grafik aus Boyer et al. (2002) , welche ein Beispiel für XML-Knoten Syntax angibt (links) und die dazugehörige Hierarchie als Baum-Datenstruktur (rechts).

Eine weitere wichtige Eigenschaft von Knoten sind Attribute. Diese werden innerhalb des Starttags oder eines leeren Tags mit name=”inhalt” initialisiert. Wobei „name“ eine eindeutige Bezeichnung innerhalb des Tags ist und „inhalt“ ein beliebiger Inhalt des Attributs ist (Boyer et al., 2002).

2.3.2 Office Open XML

OOXML ist ein von Microsoft entwickeltes Dateiformat auf XML-Basis und gilt seit 2007 innerhalb von Office Produkten wie Word, Excel und PowerPoint als Standardformat[6]. Eine OOXML-Datei ist im Prinzip ein Paket von mehreren Dateien (ECMA international, 2021). Wie in Codeauflistung 2.1 zu erkennen ist, wird eine OOXML Datei mit einem „pkg:package“ Element eingeleitet. Dieses Element gibt den Anfang und das Ende eines Paketes an. Die Elemente „pkg:part“ stellen dabei einzelne Teile, beziehungsweise Dateien des Paketes dar. Eines dieser Dateien ist die .rels Datei (Zeile 2). Sie ist besonders, da sie die Beziehungen und Funktionen einzelner Dateien angibt. In der Zeile 5 wird so zum Beispiel definiert, dass der Inhalt

[6]https://prod.support.services.microsoft.com/de-de/office/open-xml-formate-und-dateinamenerweiterungen-5200d93c-3449-4380-8e11-31ef14555b18 (Abgerufen: 10.03.2024)

vom Word Dokument in der Datei „/word/document.xml“ in der Zeile 9 beschrieben wird. Die „/word/document.xml“ Datei wiederum definert in der Zeile 15 einen formatierten Text zum Anzeigen im Word-Dokument. Es lassen sich mithilfe von OOXML auch komplexere Elemente repräsentieren wie zum Beispiel Tabellen, Graphen oder Bilder. Eine detailliertere Beschreibung des OOXML Formats lässt sich in der Spezifikation des OOXML-Standards der European Computer Manufacturers Association (ECMA) in ECMA international (2021) finden.
OOXML-Dateien mit den Dateienden .docx, .xlsx oder .pptx sind in der Regel nur von Maschinen lesbar, dennoch lassen sich Dokumente in den Office Anwendungen auch als normale XML Dateien speichern, welche den Inhalt auch für Menschen lesbar machen [7]. Diese Methodik wurde angewendet um die für die Implementierung verwendeten Templates zu erstellen.

```
1  <pkg:package xmlns:pkg="http://schemas.microsoft.com/office/2006/xmlPackage">
2    <pkg:part pkg:name="/_rels/.rels" pkg:contentType="application/vnd.openxmlformats-
         package.relationships+xml" pkg:padding="512">
3      <pkg:xmlData>
4        <Relationships xmlns="http://schemas.openxmlformats.org/package/2006/
             relationships">
5          <Relationship Id="rId1" Type="http://schemas.openxmlformats.org/
               officeDocument/2006/relationships/officeDocument" Target="word/document.
               xml"/>
6        </Relationships>
7      </pkg:xmlData>
8    </pkg:part>
9    <pkg:part pkg:name="/word/document.xml" pkg:contentType="application/vnd.
         openxmlformats-officedocument.wordprocessingml.document.main+xml">
10     <pkg:xmlData>
11         ...
12               <w:sdtContent>
13                 <w:p>
14                   <w:r>
15                   <w:t>[Example Text]</w:t>
16                 </w:r>
17                 </w:p>
18               </w:sdtContent>
19         ...
20     </pkg:xmlData>
21   </pkg:part>
22 </pkg:package>
```

Codeauflistung 2.1: Ausschnitt einer OOXML-Datei um in Microsoft Word einen einfachen Text anzuzeigen.

[7] https://learn.microsoft.com/en-us/office/dev/add-ins/word/create-better-add-ins-for-word-with-office-open-xml#see-also (Abgerufen: 10.03.2024)

2.3.3 JSON

JSON ist ein Datenformat für das Austauschen von strukturierten Daten und ist dabei unabhängig von der Plattform und Programmiersprache. Seine Syntax wurde von der Notation von JavaScript Objekten inspiriert. Seine Struktur gilt als sehr einfach, weshalb keine Änderungen an der Grammatik erwartet werden (ECMA International, 2017). Eine JSON-Datei wird ähnlich wie bei XML auch mit einem einzigen übergeordneten Objekt eingeleitet. Dieses Objekt ist ein JavaScript Objekt und wird mithilfe einer öffnenden ({) und schließenden (}) geschweiften Klammer dargestellt. Ein Objekt kann eine beliebige Anzahl von Elementen enthalten. Ein Element wird durch eine Bezeichnung in Anführungszeichen gefolgt von einem Doppelpunkt dargestellt, wie in Codeauflistung 2.2 in der Zeile 2 zu entnehmen ist. Ein Element kann den Wert eines Objektes (Zeile 7), einer Liste (Zeile 6), einer Zahl (Zeile 3), einer Zeichenkette (Zeile 2) oder die Werte „true“, „false“ und „null“ (Zeile 4 und 5) annehmen (ECMA International, 2017).

```
1  {
2      "String" : "String",
3      "Number" : 0.1,
4      "Boolean" : true,
5      "Null" : null,
6      "Array" : [0, 1, 2],
7      "Object" : {
8          "ArrayObject": [
9              {
10                 "Value": "Value"
11             }
12         ]
13     }
14 }
```

Codeauflistung 2.2: Ausschnitt einer beispielhaften JSON-Datei.

JavaScript Object Notation for Linked Data (JSON-LD) ist eine Sonderform von JSON und wurde entwickelt um Graphen zu beschreiben und Informationen gerichteter Graphen zu übertragen (Sporny et al., 2020). Es wird hierbei mit einer triple-Notation gearbeitet wie zum Beispiel auch bei dem Resource Description Framework (RDF). In dieser Triple Notation besteht jede Information aus einem Subjekt, Prädikat und Objekt. In einem Graphen würde mithilfe einer solchen Information eine gerichtete Kante von Subjekt zu Objekt mit dem Prädikat als Bezeichnung erstellt. In Codeauflistung 2.3 ist ein beispielhafter JSON-LD Code dargestellt. Hier lässt sich in der Zeile 2 ein Beispiel für ein Subjekt, welches mit dem Element „@id“ eingeleitet wird, erkennen. Alle anderen Elemente mit Ausnahme von „@context“

geben nun Prädikate an, wie zum Beispiel in der Zeile 3. Der Wert eines Prädikat-Elements ist dabei immer entweder ein konkreter Wert (Zeile 6) oder ein Objekt, welches ebenfalls Prädikate enthalten kann (Zeile 4) und somit auch gleichzeitig ein Subjekt ist. Ein Subjekt wird dabei immer als Internationalized Resource Identifier (IRI) angegeben. Dies ist eine eindeutige Referenz auf das jeweilige Objekt. Im „@context“ Element wird jeden Prädikat eine eindeutige IRI zugewiesen, wie in der Zeile 8 zu erkennen ist(Sporny et al., 2020).

```
1  {
2      "@id" : "Subjekt",
3      "Praedikat 1" : {
4          "@id" : "Objekt"
5      },
6      "Praedikat 2" : "Wert",
7      "@context": {
8          "Praedikat 1" : "https://example.com/predicate1",
9          "Praedikat 2" : "https://example.com/predicate2"
10     }
11 }
```

Codeauflistung 2.3: Ausschnitt einer beispielhaften JSON-LD Datei.

2.3.4 YAML

YAML ist eine Sprache, welche ähnlich wie JSON unabhängig von Programmiersprachen verwendet werden kann und Datentypen repräsentieren soll. YAML's Syntax ist eine Oberklasse der JSON-Syntax. Dies bedeutet, dass jede JSON-Syntax gleichzeitig ein gültiges YAML-Format ist. Des Weiteren unterstützt YAML weitere Möglichkeiten um Elemente zu deklarieren, beispielsweise können nun Tabulatoren anstelle von Klammern genutzt werden. Ein solches Beispiel ist in Codeauflistung 2.4 dargestellt. Hier kann man in der Zeile 2 und Zeile 5 den Unterschied zwischen einer Objekt Deklarierung in JSON-Syntax und mithilfe eines Tabulators erkennen. Des Weiteren kann man in der Zeile 7 auch eine Listendeklaration mithilfe von Bindestrichen sehen. Auch ersichtlich aus der Codeauflistung in der Zeile 7, ist das Semikolons in YAML im Vergleich zu JSON nicht strikt notwendig sind um Bezeichnungen oder Zeichenketten anzugeben. Mithilfe dieser erweiterten Möglichkeiten profitiert YAML von einer besseren Lesbarkeit durch den Menschen. Ein weiterer großer Unterschied zu JSON ist, dass YAML nicht nur ein einfaches Format zum angeben von Objekten ist, sondern zahlreiche komplexe Funktionen besitzt, welche beispielsweise das Definieren von Variablen in Form von „Aliases“ ermöglichen[8].

[8]https://yaml.org/spec/1.2.2/ (Abgerufen: 10.03.2024)

```
---
"json" : {
  "element" : "string",
}
yaml:
  element : string
array :
  - element 1
  - element 2
...
```

Codeauflistung 2.4: Ausschnitt einer beispielhaften YAML-Datei.

2.4 Verwandte Arbeiten

Das Konzept dieser Arbeit baut auf einer Reihe von anderen wissenschaftlichen Arbeiten auf. Diese werden nun in diesem Abschnitt vorgestellt, dabei werden auch ähnliche Systeme, an welchen sich das zu implementierendes System orientiert hat, beschrieben. In Abschnitt 2.4.1 wird auf das LATEX-Paket SciKGTeX, welches konzeptionelle Ähnlichkeiten zu der entwickelten Software aufweist, eingegangen. In Abschnitt 2.4.2 folgt eine Beschreibung der Python und R Bibliothek vom ORKG, welche besonders wichtig für die Generierung der maschinenlesbaren Daten ist. Anschließend folgt in Abschnitt 2.4.3 eine weitere Vorgehensweise um Daten im ORKG einbinden zu können.

2.4.1 SciKGTeX

SciKGTeX ist ein LATEX-Paket, welches es ermöglicht, relevante Information innerhalb der Schreibumgebung zu annotieren. Diese Informationen können dann für den ORKG in Form eines Beitrags bereitgestellt werden. Bei der Verwendung des Paketes in einer Latex Umgebung ist es demnach für einen Benutzer möglich relevante Daten innerhalb der Quelldateien zu markieren. Diese Annotationen werden danach bei der PDF-Generierung in Form von Metadaten in die PDF eingefügt (Bless, 2022). Anschließend ist es für den ORKG möglich diese strukturierten Daten zu extrahieren und in Form eines Beitrages dem Wissensgraph hinzufügen.
Dieses Paket stellt aufgrund ihrer zahlreichen Ähnlichkeiten eine große Inspirationsquelle für dieses Arbeit dar. Beide Technologien gelten als Anreiz zur Erstellung von Beiträgen zum ORKG und kommen während des Schreibvorgangs zum Einsatz. Anders als SciKGTeX liegt der Anreiz des umzusetzenden Systems dennoch an einer

automatischen Visualisierung von Daten, welche die Templates vom ORKG nutzt.

2.4.2 ORKG Python- und R-Bibliothek

Ein wichtiger Bestandteil vom ORKG ist die Python[9] und R (Boubakri, 2022) Bibliothek. Diese Bibliotheken ermöglichen den Nutzern die ORKG-Templates und Daten innerhalb einer Python oder R Umgebung zu nutzen und hat das Ziel diese Daten auch für Maschinen verwendbar zu machen. Abgesehen vom eignen Nutzen an strukturierten Daten, zur Verwendung innerhalb digitaler Systeme, ist es auch vorteilhaft diese Daten und dessen Programmcodes parallel zu seinem schriftlichen Artikel zu veröffentlichen. Somit wird gewährleistet, dass Daten wiederverwendbar gemacht werden und andere Forscher diese für weitere Forschungen oder Tests verwenden können.
Eine besonders wichtige Funktion der Bibliotheken für den Rest dieser Arbeit ist das materialisieren von Templates. Bei dieser Funktion geht es darum, Code dynamisch abhängig von ORKG-Templates zu generieren (Boubakri, 2022). Beispielsweise hat man einen Datensatz mit zwei Variablen. Anhand der Daten will man nun einen P-Wert ermitteln, welcher den Grad eines statistischen Zusammenhangs zwischen den beiden Variablen angeben soll. Für dieses Beispiel könnte das „Student's t-test“ Template verwendet werden. Hierzu ist in Codeauflistung 2.5 ein Python Code gegeben, welcher für einen Datensatz „dataset“ unter Verwendung des „Student's t-test“ Templates eine JSON-LD Datei generiert. In der Zeile 4 wird das Template materialisiert. Der Parameter „R12002“ ist dabei die Identifikation (ID) des Templates. Somit wird nun in der Zeile 7 ein Objekt des Templates erstellt, welches mit den Ergebnissen der Auswertung des Datensatzes gefüllt wird. In der Zeile 9 wird die IRI für die gemessene Größe beider Parameter angegeben. Zeile 10 definiert den verwendeten Datensatz „dataset“ und Zeile 11 gibt den P-Wert als das Ergebnis der Untersuchung an. Anschließend wird in der Zeile 14 aus den Daten und dem Format des Templates eine JSON-Datei erstellt[10]. Diese Ausgabedatei kann nun für weitere Anwendungen genutzt werden.

2.4.3 Datenerfassung nach der Veröffentlichung

Wie schon in Abschnitt 2.4.2 erwähnt, kann es hilfreich sein neben einem schriftlichen Dokument auch dazugehörige maschinenlesbare Daten zu veröffentlichen, beispielsweise als JSON-LD Dateien. Damit Veröffentlichungen auch noch nach der

[9] https://orkg.readthedocs.io/en/latest/ (Abgerufen: 10.03.2024)
[10] https://orkg.org/help-center/article/47/Machine_reusable_by_design (Abgerufen: 10.03.2024)

```python
from orkg import ORKG, Hosts

orkg = ORKG(host=Hosts.PRODUCTION)
orkg.templates.materialize_template("R12002")
tp = orkg.templates

tp.students_ttest(
  label="Statistically significant hypothesis test",
  has_dependent_variable= uri,
  has_specified_input=(dataset, "a example dataset"),
  has_specified_output=tp.pvalue("the p-value",
    tp.scalar_value_specification("{}".format(str(pvalue)), str(pvalue))
  ),
).serialize_to_file("output.json", format="json-ld")
```

Codeauflistung 2.5: Python Code für ORKG-Template Materialisierung und Verwendung.

Veröffentlichung leicht vom ORKG profitieren können, wurde eine weitere Möglichkeit zur Erstellung von Beiträgen entwickelt. Bei dieser Methode muss ein Dokument mit seinen Daten erst mit einer eindeutigen Digital Object Identifier (DOI) versehen sein (Anfuso, 2023). Eine DOI ist eine eindeutige digitale Identifikation von physischen oder digitalen Objekten (ISO, 2022). Diese DOI gilt nun als Referenz zu dem Dokument und einiger JSON-LD Dateien. Diese JSON-LD Dateien können beispielsweise mithilfe der Python oder R Bibliothek vom ORKG erzeugt werden, wie bereits in Abschnitt 2.4.2 beschrieben. Anschließend lässt sich der Beitrag automatisch durch Angabe der DOI auf der Website vom ORKG einbinden (Anfuso, 2023).

Kapitel 3

Ansatz

In diesem Kapitel geht es um die Beschreibung des Systemansatzes und seiner Ziele. Zuerst wird das zu implementierende System nur als Blackbox behandelt. Anschließend wird der genannte Ansatz mit einer Implementierung vorgestellt, für die später auch eine Testausführung präsentiert wird. In Abschnitt 3.1 wird zunächst ein Überblick über die Umgebung des Systems gegeben. Dabei wird ein beispielhafter Ablauf beschrieben, der verdeutlicht, wie das System im Schreibprozess wissenschaftlicher Arbeiten verwendet wird. Anschließend folgt im Abschnitt 3.2 eine Beschreibung der Ziele und Anforderungen an das System. In Abschnitt 3.3 werden verschiedene Möglichkeiten zur Umsetzung vorgestellt, wobei insbesondere auf die Unterschiede der möglichen zugrundeliegenden Plattformen eingegangen wird. Als nächstes wird im Abschnitt 3.4 die Implementation vorgestellt, welche im Rahmen dieser Arbeit entwickelt wurde. Abschnitt 3.5 beschreibt danach die Vorgehensweise und die Auswertung einer Testausführung.

3.1 Einordnung innerhalb des ORKG Systems

Nun folgt eine Beschreibung der Interaktion aller beteiligter Systeme, welche für das umzusetzende System erforderlich sind. Dabei geht man von einem Anwender aus, welcher seine Resultate mithilfe des Angebots vom ORKG visualisieren möchte. Dazu ist in Abbildung 3.1 ein vollständiger Ablauf in mehreren Schritten dargestellt. Für den Anfang wird davon ausgegangen, dass der Nutzer die Daten für eine Visualisierung bereits gesammelt hat und nun mit dem schreiben eines wissenschaftlichen Artikels beginnen will. Die einzelnen Schritte eines Ablaufs werden nun der Reihe nach beschrieben:

1. Sobald der Nutzer seine Resultate gesammelt hat, ist es für ihn möglich seine

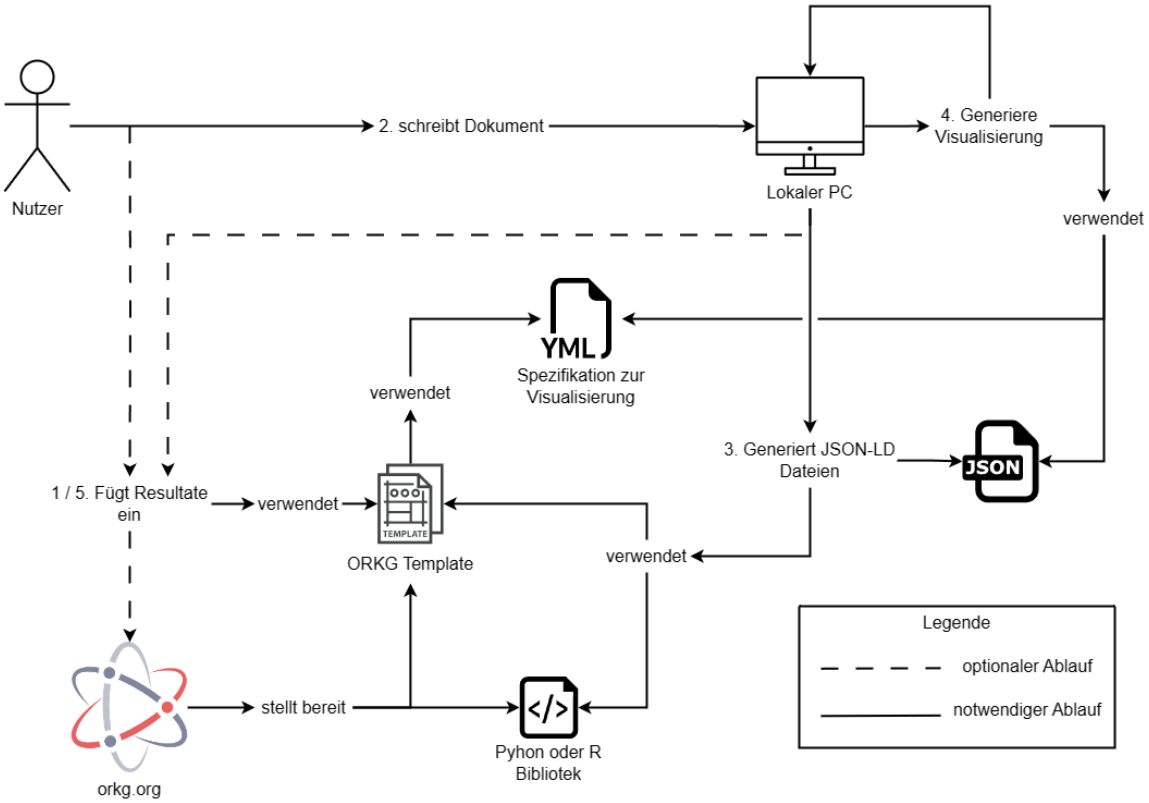


Abbildung 3.1: Ablauf zur Nutzung des zu implementierenden Programms mit den ORKG Funktionalitäten.

gesammelten Daten über die Website vom ORKG einzubinden. Dieser Schritt ist jedoch optional und kann jederzeit in späteren Schritten durchgeführt werden[1].

2. Anschließend beginnt der Nutzer mit dem Schreiben seiner wissenschaftlichen Arbeit in einer Textverarbeitungssoftware auf seinem Computer. Nun kann er die Struktur seines Artikels planen und überlegen, an welcher Stelle er eine bestimmte Visualisierung einbinden möchte. Dazu muss der Nutzer, das in dieser Arbeit vorgestellte Programm oder Erweiterung installieren, beziehungsweise aktivieren.

3. Sobald der Nutzer seine Ergebnisse visualisieren möchte, muss dieser die in Abschnitt 2.4.2 beschriebene Python oder R Bibliothek installieren und mithilfe

[1]https://orkg.org/ (Abgerufen: 10.03.2024)

dieser die erforderlichen JSON-LD Dateien generieren. Das genaue Vorgehen zum generieren der Dateien hängt stark davon ab, ob bei der Ermittlung der Daten bereits eine Python oder R Umgebung verwendet wurde.

4. Nachdem die JSON-LD Dateien generiert wurden, kann der Nutzer nun mithilfe der installierten, beziehungsweise aktivierten Software die Visualisierungen durchführen. Die Software nimmt die JSON-LD Dateien als Eingabe entgegen und generiert abhängig von den gewählten Templates eine geeignete Visualisierung in der jeweiligen Umgebung der Textverarbeitungssoftware. Das genaue Format für die Visualisierung wird dabei durch eine vom Template abhängige Spezifikation angegeben. Sobald die Dateien von der Software verarbeitet wurden und eine Visualisierung dargestellt wurde, kann der Nutzer bei Bedarf wieder zu Schritt 3 oder 2 zurückkehren und weitere Daten auswerten lassen.

5. Am Schluss, wenn alle Daten durch die Software visualisiert wurden, kann der Artikel zu Ende geschrieben werden. Anschließend hat der Nutzer die Möglichkeit seine gesammelten maschinenlesbaren Daten parallel mit seinem Schreiben zu veröffentlichen. Dieser Schritt erlaubt nochmals eine Anbindung der Daten ans ORKG wie in Abschnitt 2.4.3 beschrieben wurde, wenn Schritt 1 nicht durchgeführt wurde. Zudem ermöglicht dieser Schritt auch die Wiederverwendbarkeit der gesammelten Ergebnisse.

3.2 Aufgabe und Ziele des Systems

Das eigentliche System kommt in Schritt 4 in Abbildung 3.1 zum Einsatz. Die Aufgabe des Systems ist es eine grafische Oberfläche anzuzeigen, auf welcher ein Benutzer eine beliebige Anzahl an JSON-LD Dateien angeben kann. Für diese Dateien soll dann eine respektive Visualisierung generiert werden.
Für die Entwicklung einer Anwendung müssen zudem einige Anforderungen definiert werden, welche innerhalb der Implementierung gewährleistet werden müssen. In der Nachfolgenden Auflistung werden einige wichtige Ziele genannt und leicht nach der Priorität sortiert.

1. Darstellung fehlerfreier Daten
2. Korrekte Darstellung von Templates
3. Einfache Verwendung der Anwendung
4. Unterstützung zahlreicher Visualisierungen

5. Vermeidung von Nachbesserungen des Nutzers
6. Einfache Installation oder Aktivierung der Anwendung
7. Zusätzliche Funktionen (Untertitel, Maximalwert, usw.)

3.3 Systemumsetzung

Es gibt eine Vielzahl von Möglichkeiten ein derartiges System umzusetzen, dabei gibt es dennoch auch eine Reihe von Faktoren welche zu beachten sind. In diesem Abschnitt werden einige zur Wahl stehenden Möglichkeiten für die Implementierung genannt und hinsichtlich ihrer Vorteile und Nachteile bewertet.

3.3.1 Interaktion mit der Schreibumgebung

Eine wichtige Eigenschaft für ein umzusetzendes System ist die Art der Interaktion mit einer Schreibumgebung. In Abbildung 3.2 sind zwei unterschiedlichen Konzepte dargestellt.
Das erste Konzept einer Entkoppelten Software (Vergleiche 3.2 links) beschreibt ein System, welches unabhängig von einer Textverarbeitungssoftware ausgeführt wird, also separat gestartet wird. Der Benutzer muss dann die Ausgabe des Systems manuell in das Dokument einfügen. Es besteht die Möglichkeit die Visualisierungsobjekte als plattformunabhängiges Format, wie zum Beispiel PNG, JPEG oder SVG, auszugeben, um die Visualisierungen auf jedmögliche Plattform zu unterstützen. Ein spezielleres Format kann dennoch Lohnenswerter sein, da solche Visualisierungen auf der jeweiligen spezifischen Plattform zusätzliche Funktionen erlauben können, wie zum Beispiel das Anpassen und Editieren einzelner Elemente.
Der zweite Ansatz (Vergleiche 3.2 rechts) beschreibt eine Erweiterung für eine Textverarbeitungssoftware, welche eine automatische Anbindung der Visualisierung im Dokument erlaubt.
Beide Ansätze haben ihre Vor- und Nachteile, beispielsweise haben entkoppelte Anwendungen den Vorteil, dass dessen Funktionalitäten nicht von der unterliegenden Anwendung beschränkt werden. Ein weiterer Vorteil ist auch die Freiheit bei der Wahl der Technologien. Während man bei Erweiterungen nur eine bestimmte Menge von Programmiersprachen, Bibliotheken und sonstiges zur Auswahl hat, wird eine separate Software lediglich durch das Betriebssystem und jeweiliger Laufzeitumgebungen limitiert. Erweiterungen sind dennoch in der Regel leichter zu installieren, da vieles oft von der unterliegenden Anwendung gesteuert wird. Des Weiteren lassen

sich mit einer Erweiterungen Dokumente direkt bearbeiten, sodass sich der Benutzer nicht um das Einfügen von Elementen in ein Dokument kümmern muss.
Neben den beiden genannten Konzepten ist es auch möglich eine entkoppelte Anwendung zu erstellen, welche das zu bearbeitende Dokument als Eingabe nimmt und die jeweiligen Visualisierungen erstellt. Da Dokumente von einigen Textverarbeitungssoftwares ohne enormen Aufwand nur schwer in externen Programmen dargestellt werden können, muss bei diesen Anwendungen auf eine Darstellung der Dokumente verzichtet werden. Daraus folgt auch das Funktionen wie die Bestimmung der Position einzelner Visualisierungen in diesen Dokumenten nur sehr schwer zu definieren sind. Aus diesem Grund ist ein solcher Ansatz oft nur sehr schwer möglich und wird demnach im weiteren nicht behandelt.

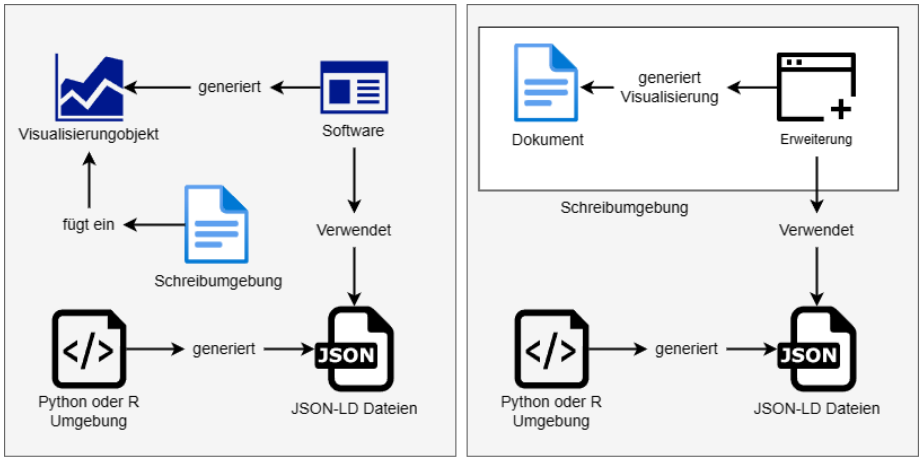


Abbildung 3.2: Möglichkeiten zur Umsetzung der Interaktion von Anwendung mit der Schreibumgebung. Dargestellt sind die Interaktion einer entkoppelte Software (links) und einer Erweiterung (rechts).

3.3.2 Wahl der Textverarbeitungssoftware

Es gibt zahlreiche Textverarbeitungssoftwares welche als Zielplattformen für die Visualisierungen zur Auswahl stehen. Im Folgenden werden übersichtshalber dennoch nur drei dieser Plattformen betrachtet: Microsoft Word, Google Docs und eine beliebige Latex-Umgebung. Diese wurden aufgrund ihrer Popularität und ihrer starken technischen Unterschiede gewählt. Wie bereits im Abschnitt 2.2 beschrieben wurde, ist es in Microsoft Word möglich weitere Funktionalitäten mithilfe von JavaScript in Form eines Add-Ins zu implementieren. Google Docs unterstützt eine JavaScript

Plattform namens App Script, mit welcher Add-ons definiert werden können[2]. Latex selber ist anders als die beiden anderen keine Plattform sondern ein Softwarepaket. Stattdessen bilden einzelne Latexumgebungen, wie zum Beispiel Overleaf[3] eine Plattform. aus diesem Grund ist das Schreiben einer Erweiterung für den Fall Latex nicht möglich, da sich Latexumgebungen stark voneinander unterscheiden können. Das Schreiben eines Dokuments mit Latex ähnelt dem Programmieren, da man Formatierungen, Abbildungen und weiteres mit besonderen Code generiert, anstatt diese mit einzelnen Menüs auf einer grafischen Oberfläche einzufügen[4]. Eine Umsetzung für Latex lässt sich mit einer separaten Software durchführen, welche entweder einzelne Codestücke generiert oder das Dokument direkt bearbeitet. Die Gründe warum eine Implementierung in Microsoft Word vorteilhaft ist, sind die Verbreitung der Plattform und die Unterstützung von Add-Ins in Form von gehosteten Websites. Google Docs bietet ebenfalls das Erstellen von Add-Ons an und wird anders als Microsoft Word kostenlos angeboten. Latex bietet eine umfangreichere Anpassung, da Latex-Dokumente flexibler sind und es eine Vielzahl an erweiternden Bibliotheken existieren. Der Nachteil an Latex ist dennoch, dass Latex aufgrund der Schreibweise von Dokumenten deutlich komplizierter ist.

3.4 Technische Umsetzung

Für die technische Umsetzung wurde die Textverarbeitungssoftware Microsoft Word von Microsoft aufgrund ihrer starken Verbreitung als Plattform für die Implementierung gewählt. Wie bereits im Abschnitt 3.3.2 beschreiben ist es demnach sinnvoll die Software in Form eines Add-Ins zu implementieren. Das Add-In ist eine React-Web Applikation, welche auf einem Server gehostet wird. Im Entwicklungsprozess wurde dieser Server lediglich lokal ausgeführt. Für eine großflächigere Bereitstellung, wie zum Beispiel auf dem internen Add-In Store von Microsoft, müsste ein seperater Server zur Verfügung stehen, welcher auf Anfragen einzelner Clients den Code der Web-Applikation bereitstellen würde. Wichtig für die Benutzung des Add-Ins ist in der Regel nur das sogenannte Manifest. Dieses ist eine XML-Datei, welche die notwendigen Informationen für Word enthält, um das Add-In von dem jeweiligen Server zu laden[5]. Die im Abschnitt 3.1 genannten Spezifikationen zur Visualisierung wurden als YAML-Spezifikationen umgesetzt und für die Umsetzung vorerst lokal

[2]https://developers.google.com/apps-script?hl=de (Abgerufen: 10.03.2024)

[3]https://www.overleaf.com/ (Abgerufen: 13.03.2024)

[4]https://www.latex-project.org/ (Abgerufen: 10.03.2024)

[5]https://learn.microsoft.com/en-us/office/dev/add-ins/publish/publish-add-in-vs-code (Abgerufen: 10.03.2024)

gespeichert.
Der in Typescript geschriebene funktionale Programmcode lässt sich in mehrere Schritte unterteilen, die jeweils eine andere Aufgabe erfüllen und nacheinander ausgeführt werden. Die Schritte werden in Abbildung 3.3 dargestellt und im weiteren erläutert:

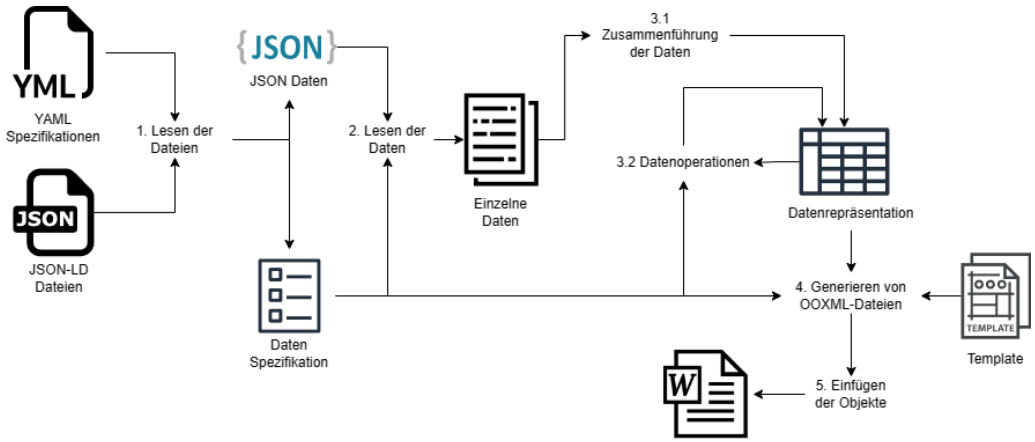


Abbildung 3.3: Kommentar auf die Versionsüberprüfung von Word bei Benutzung von Fluent UI React.

1. Lesen der Dateien: Die JSON-LD und YAML-Dateien werden gelesen und gesammelt. Die Codeauflistung 3.1 zeigt ein Beispiel für eine YAML-Spezifikation eines jeweiligen ORKG-Templates. Im Weiteren wird die Struktur dieser Spezifikationen nun näher beschrieben und erklärt wie diese Informationen helfen eine passende JSON-LD Datei zu lesen.
   Der in der Zeile 2 angegebene Parameter „type“ gibt die Adresse des ORKG-Templates an um diesen eindeutig identifizieren zu können. Codeauflistung 3.2 zeigt ein Ausschnitt einer JSON-LD Datei vom selben Typ, erkennbar anhand der gleichen Adresse als Wert für den Parameter „type“. Der Parameter „structure“ (Zeile 3) gibt die Struktur der Dateien an. Die Elemente im „structure“ Parameter geben den Namen des Pfades in einem JSON-LD Dokument an. So kann man jedes Element im „structure“ Parameter auch in einem passenden JSON-LD Dokument finden. Demnach kann man beispielsweise erkennen, dass sich in der JSON-LD Datei auch ein Parameter „label“ befinden muss (Zeile 3 in Codeauflistung 3.2), da dieser in der Zeile 4 der YAML-Datei innerhalb des „structure“ Parameters enthalten ist.

2. Lesen der Daten: Die in den JSON-Dateien enthaltenen Daten werden nacheinander mithilfe der zugehörigen YAML-Beschreibung gelesen. Ein Parameter in einer JSON-Datei besitzt Daten, wenn für dessen Pfad in der YAML-Spezifikation ein „datatype“ Parameter existiert (Zeile 5 in Codeauflistung 3.1). Dieser Parameter weist der Information einen Datentyp zu. Für gewöhnlich werden die Daten in Form von Zeichenketten (Strings) gespeichert. Der Parameter „data“ (Zeile 7) folgt immer nach einem „datatype“ Parameter und gibt an, welche Informationen aus einem Datenwert zu entnehmen sind. Zum Beispiel gibt der „data“ Parameter in Codeauflistung 3.1 an, dass die Daten „Column“, „Table“ und „Row“ aus dem jeweiligen Datenwert zu entnehmen sind. Wie man anhand dieses Beispiels sehen kann, können auch mehrere Informationen aus einem Element entnommen werden. In diesem Beispiel sind diese in einer Zeichenkette gespeichert, welche die einzelnen Daten durch einen im Parameter „seperator“ angegebenen Trennzeichen (Zeile 6) voneinander trennt. Eine solche Speicherung ist dabei eine Konvention und ist für diesen Typen von Template erforderlich.

3. Zusammenführung von Daten: In diesem Schritt werden zusammengehörige Daten gesammelt und in eine Zwischenform gebracht. Ausschlaggebend für diese Zwischenform ist die Form der Visualisierung. Derzeit ist die Anwendung nur für tabellarische Daten ausgelegt, weshalb für die Daten immer zwei dimensionalen Listen erstellt werden. Anschließend können auf diesen Listen weitere Operationen angewendet werden, wie zum Beispiel Sortierungen, Wertetransformationen oder andere Spalten-, Zeilen- und Werteoperationen.

4. Generieren von OOXML-Dateien: Mithilfe der erzeugten strukturierten Daten und eines Templates werden nun einzelne OOXML-Dokumente erzeugt. Das passende Template wird dabei durch den „visualization“ Parameter ausgewählt (Zeile 21). Für die Templateerstellung wurde die Template-Engine Handlebars[6] weshalb die Templates als Handlebars-Dateien vorliegen.

5. Einfügen der Objekte in das Word Dokument: Die erzeugten OOXML-Dateien werden mithilfe der Javascript API in das offene Word Dokument geschrieben.

[6]https://www.npmjs.com/package/handlebars (Abgerufen: 10.03.2024)

```yaml
---
type: "https://orkg.org/class/C17002"
structure:
  label:
    datatype: "string"
    seperator: '/'
    data:
      - Column
      - Table
      - Row
    role: "data"
  PWC_HAS_BENCHMARK:
    P34:
      P45073:
        P45075:
          datatype: "string"
          data: Value
          role: "data"
details: {
  multiple_files: true,
  visualization: "Table"
}
...
```

Codeauflistung 3.1: YAML Spezifikation für ein beispielhaftes ORKG-Template.

```json
{
    "@id": "_:n1",
    "label": "ORKG/SciBERT/1",
    "@type": [
        "https://orkg.org/class/C17002"
    ],
    "P1004": [
        {
            "@id": "https://orkg.org/resource/R125989"
        }
    ],
...
```

Codeauflistung 3.2: Ausschnitt einer beispielhaften JSON-LD Datei für die Implementierung.

3.5 Evaluation

Im Rahmen des Projektes wurden drei Use Cases, mit welchen das entwickelte Programm getestet wurde, gewählt. In diesem Abschnitt geht es um eine konkrete Ausführung des Programms um einen beispielhaften Programmablauf darzustellen. Im Abschnitt 3.6 werden die Use Cases näher erläutert, anschließend wird in Abschnitt 3.7 ein Testlauf beschrieben, bei dem die genannten Daten der Use Cases als Eingabe verwendet werden. Zuletzt wird die Testauswertung im Abschnitt 3.8 mit den Referenzdarstellungen der Use Cases verglichen und bewertet.

3.6 Use Cases

In diesem Abschnitt werden nun die Use Cases beschrieben. Der erste Use Case ist das Generieren eines Box-Plot Diagramms. Der verwendete Datensatz stammt von der Arbeit Haddad et al. (2016) und lag in Form einer JSON-LD Datei bereit. Die ORKG-Daten sind eine Instanz des „Student's t-test"-Template[7]. Mit dem Diagramm soll inhaltlich dargestellt werden, inwiefern sich eine Bindungsaktivität eines auf Eisen reagierenden Elements im linksventrikulären Gewebe bei versagenden und nicht-versagenden Herzen unterscheidet. Das Ziel diesen Vergleichs ist es zu zeigen, dass es einen statistisch signifikanten Unterschied zwischen versagenden und nicht-versagenden Herzen gibt (Haddad et al., 2016).
Der zweite Use Case ist eine weitere Instanz des „Student's t-test" Templates, für welchen ebenfalls eine Kastengrafik dargestellt werden soll. In diesem Beispiel geht es um die Visualisierung von Längen des Blütenblattes zweier Pflanzenarten. Das genutzte Datensatz für die Datenwerte ist das Iris-Datensatz[8] und ist ein weit verbreitetes Beispiel im Bereich des maschinellen Lernens oder in der Statistik (Unwin & Kleinman, 2021)
Für den dritten Use Case müssen mehrere Tabellen generiert werden. Die Daten aus Thießen et al. (2023) müssen aus mehrere JSON-LD Dateien gelesen und anschließend kombiniert werden. Das verwendete Template ist das „Leaderboard"-Template[9]. Dabei geht es in diesem Use Case darum, für jeweils vier Large Language Models (LLMs) eine Übersicht von F1-Scores für kMeans Clustering-Algorithmen auf vier verschiedenen Datensätze, darzustellen (Thießen et al., 2023). Für jedes dieser LLMs soll demnach eine eigne Tabelle mit jeweils vier Spalten für jeden Datensatz generiert werden. Zusätzlich können einige LLMs mehrere Ebenen (Layers) besitzen, welche

[7]https://orkg.org/template/R12002/properties?isEditMode=false (Abgerufen: 10.03.2024)
[8]https://archive.ics.uci.edu/dataset/53/iris (Abgerufen: 10.03.2024)
[9]https://orkg.org/template/R107801/properties?isEditMode=false (Abgerufen: 10.03.2024)

die Zeilen der Tabellen darstellen. Das Ziel hierbei ist es die LLMs, sowie einzelne Layers hinsichtlich ihrer Leistung auf unterschiedlichen Datensätze vergleichen zu können.

3.7 Testausführung

Sobald das Add-In in Word geladen wurde, erhält man, dass in Abbildung 3.4 dargestellte Fenster rechts neben der Dokumentanzeige. Sollte sich rechts kein geöffnetes Fenster befinden, dann sollte es möglich sein diesen mit einem Klick auf der Schaltfläche mit dem ORKG Logo und dem Text „Show Taskpane“ zu öffnen. Beim Klicken

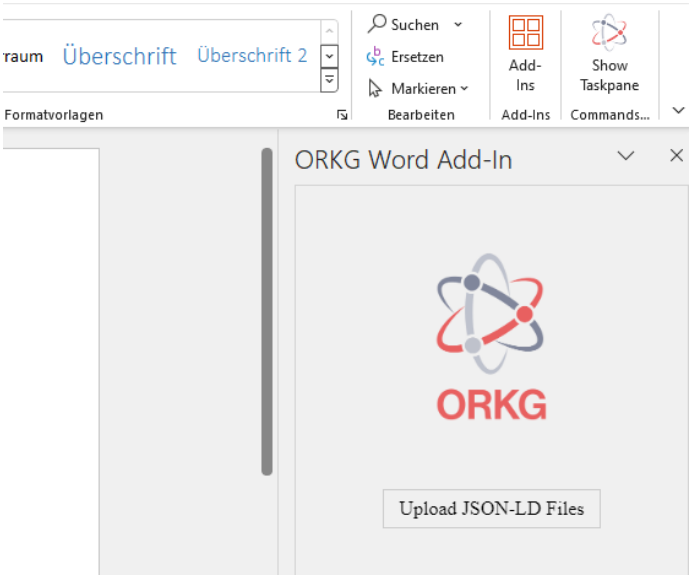


Abbildung 3.4: Geöffnetes ORKG Add-In Fenster in Word.

des Knopfes „Upload JSON-LD Files“ öffnet sich ein Dateiexplorer in einem separaten Fenster, in welchen es möglich ist mehrere JSON-Dateien auszuwählen. Nach dem Bestätigen von einer oder mehreren JSON-Dateien werden diese, wie bereits im Abschnitt 3.4 beschrieben, verarbeitet, sodass eine Visualisierung im Dokument ausgegeben wird. Bei Eingabe der JSON-LD Dateien beider Use Cases werden die in Abbildungen 3.5A, 3.6 und 3.7A dargestellten Elemente generiert.

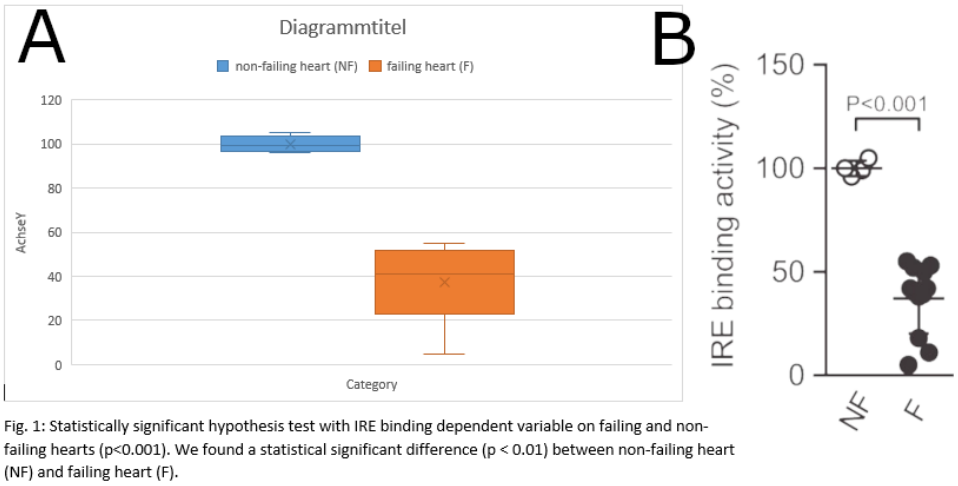


Fig. 1: Statistically significant hypothesis test with IRE binding dependent variable on failing and non-failing hearts (p<0.001). We found a statistical significant difference (p < 0.01) between non-failing heart (NF) and failing heart (F).

Abbildung 3.5: Vergleich einer generierten Kastengrafik (A) mit einer Referenz-Visualisierung (B) des „Student's t-test“ Use Case aus Haddad et al. (2016) in Microsoft Word.

3.8 Testauswertung

Als Referenz für die Add-In Visualisierungen werden nun im Weiteren die in den Arbeiten Thießen et al. (2023) und Haddad et al. (2016) verwendeten Diagramme der selben Daten bezogen. Diese sind in den Abbildungen 3.5B und 3.7B abgebildet. Bei einem Vergleich beider Tabellen (Abbildung 3.7) lässt sich feststellen, dass sich einige Spaltenbezeichnungen leicht unterscheiden. Beispielsweise erhält man für die dritte Spalte die Bezeichnung „SemEval-17“ anstelle von „SemEval 2017“. Auch die Oberbezeichnung „dataset“ wurde mit der Bezeichnung „category“ nicht korrekt generiert. Ähnlich unterscheiden sich auch die Zeilennamen. In der Tabelle aus Thießen et al. (2023) wird jede Zeilenbezeichnung mit „layer“ begonnen während diese in der generierten Version fehlt.
Das generierte Diagramm unterscheidet sich optisch stark vom Referenzdiagramm (Vergleiche die Diagramme A und B in Abbildung 3.5). Die Verteilung der Datenwerte scheint jedoch die Gleiche zu sein. Auch in diesem Beispiel werden einige Bezeichnungen wie zum Beispiel „AchseY“ nicht korrekt betitelt. Für das Iris Datensatz lag keine Referenzabbildung vor, dennoch kann man auch hier die selben Probleme von fehlenden Bezeichnungen feststellen. Interessant an den generierten Diagrammen

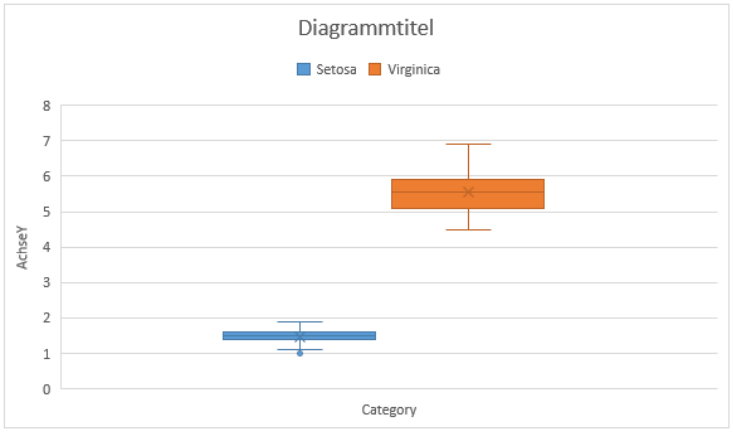


Fig. 1: Statistically significant hypothesis test with petal length dependent variable on setosa and virginica irises. We found a statistical significant difference ($p < 0.01$) between Setosa and Virginica.

Abbildung 3.6: Kastengrafik Visualisierung des „Student's t-test“ Use Case des Iris Datensatzes in Microsoft Word.

sind die automatisch generierten Untertitel, welche auf die statistische Auswertung eingehen.
Eine Wichtige Eigenschaft der Visualisierungen, welche anhand der Abbildungen nicht zu erkennen ist, ist das die generierten Objekte als Word Objekte eingefügt werden. Demnach lassen sich die Tabelle und das Diagramm nach dem Einfügen vollständig verändern. Im Diagramm lassen sich beispielsweise einzelne Teile löschen oder ändern. So ist es zum Beispiel möglich auf das Label „Category“ zu klicken und diesen mit dem drücken der Taste „entf“ zu löschen.

	category			
	ORKG	CS-KG	SemEval-17	SciERC
1	0.618	0.789	0.573	0.668
2	0.604	0.784	0.572	0.651
3	0.611	0.765	0.568	0.65
4	0.597	0.752	0.559	0.642
5	0.605	0.803	0.565	0.644
6	0.597	0.874	0.562	0.663

Table 1: BART

A: Tabellen Visualisierung für eine LLM (BART) des „Leaderboard“ Use Case in Microsoft Word

BART layer	*dataset*			
	ORKG	CS-KG	SemEval 2017	SciERC
layer 1	**0.618**	0.789	**0.573**	**0.668**
layer 2	0.604	0.784	0.572	0.651
layer 3	0.611	0.765	0.568	0.65
layer 4	0.597	0.752	0.559	0.642
layer 5	0.605	0.803	0.565	0.644
layer 6	0.597	**0.874**	0.562	0.663

B: Tabelle aus Thießen et al. (2023) als Referenz für die Visualisierung

Abbildung 3.7: Vergleich einer generierten Tabelle (A) und der Referenztabelle (B) aus Thießen et al. (2023) .

Kapitel 4

Diskussion

In diesem Kapitel werden die Ziele und Hoffnungen an das entwickelte Programm aufgegriffen und erläutert, inwiefern diese gelungen sind oder nicht erfüllt werden konnten. Des Weiteren wird reflektiert, auf welcher Weise erkannte Probleme in einer alternativen Umsetzung hätten vermieden werden können, beziehungsweise wie diese erfolgreich behandelt wurden. Dazu werden im Abschnitt 4.1 die während der Entwicklung aufgetretenen Probleme geschildert. Anschließend werden im Abschnitt 4.2 die positiven Aspekte der Umsetzung genannt und hinsichtlich der Probleme bewertet. Im Abschnitt 4.3 werden die Schwächen der Implementierung dargestellt und mögliche Lösungsmaßnahmen genannt. Auch die Wahl einer alternativen Plattform wird dabei in Betracht gezogen und mit anderen möglichen Problemen verglichen. Als letztes wird in Abschnitt 4.4 die Forschungsfrage nochmals aufgegriffen und beantwortet.

4.1 Probleme bei der Implementierung

Während der Entwicklung des bereits beschriebenen Systems (Abschnitt 3.4) sind eine Reihe von Problemen aufgetreten. Einige sehr schwierige Probleme werden nun im Genaueren geschildert:

4.1.1 Versionsunterschiede und Inkompatibilität

Aufgrund dessen, dass die Tests anfangs nur auf der Version „Word 2021“ durchgeführt wurden, konnten einige größeren Problemquellen erst sehr spät im Entwicklungsprozess erfasst werden. Einer dieser Probleme war die Verwendung von Fluent

UI React. Fluent UI React ist ein Open-Source React Front-End Framework, welches entwickelt wurde um den Designstil von Anwendungen an denen von Microsoft anzupassen[1]. Das Paket wurde bei der Initialisierung des Word Add-Ins automatisch vom Yeoman-Generator hinzugefügt.
Der Grund für die Inkompatibilität lässt sich in einem Teil des automatisch generierten Codes finden: Vor dem Laden der grafischen Oberfläche der Anwendung wird ein Skript ausgeführt, welches bei den Versionen unterhalb von Word 2021, die in Abbildung 4.1 dargestellte Fehlermeldung ausgibt. Zusätzlich wird ein Kommentar im

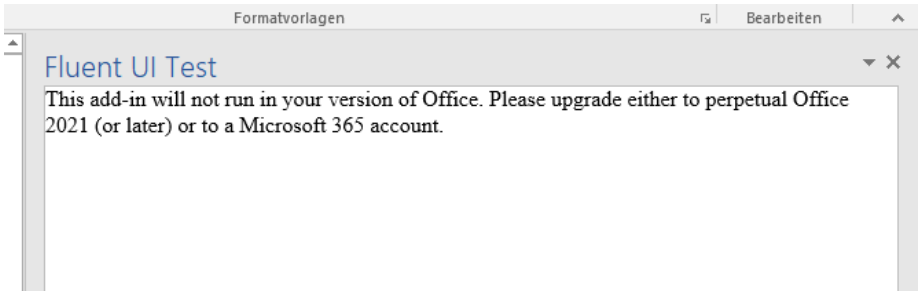


Abbildung 4.1: Fluent UI React Fehlermeldung.

Code hinterlegt, welcher den Grund für die Versionsüberprüfung näher begründet. Wie der in Abbildung 4.2 dargestellte Kommentar beschreibt, lässt sich Fluent UI React nicht auf älteren Versionen von Word ausführen, da diese veraltete Technik zur Anzeige von HTML verwenden. Auch ein Wechsel auf ältere Versionen oder alternativen Frameworks erwies sich als problematisch.

```
<!--
    Fluent UI React v. 9 uses modern JavaScript syntax that is not supported in
    Trident (Internet Explorer) or EdgeHTML (Edge Legacy), so this add-in won't
    work in Office versions that use these webviews. The script below makes the
    following div display when an unsupported webview is in use, and hides the
    React container div.
-->
<div id="tridentmessage" style="display: none; padding: 10;">
    This add-in will not run in your version of Office. Please upgrade either to
    perpetual Office 2021 (or later) or to a Microsoft 365 account.
</div>
```

Abbildung 4.2: Kommentar auf die Versionsüberprüfung von Word bei Benutzung von Fluent UI React.

Ein weiteres großes Problem war das Finden von Funktionen der JavaScript APIs,

[1]https://developer.microsoft.com/en-us/fluentui#/get-started/web (Abgerufen: 19.02.2024)

welche auch weiterhin auf älteren Versionen funktionieren. Beispielsweise wurde zuvor die Methode „insertTable“ verwendet, die im API-Set „WordApi 1.3“ enthalten ist und somit erst ab Word 2019 unterstützt wird[2]. Während man die soeben genannte Funktion gut ersetzen konnte, gab es auch Fälle in denen kompatible Funktionen andere große Probleme mit sich brachten und somit nicht verwendet werden konnten. Einer dieser Methoden ist die „setSelectedDataAsync“ Methode, welche die Auswahl eines Dokuments durch ein neues Objekt überschreibt. Diese Funktion lässt sich auf allen Versionen von Word mit JavaScript Kompatibilität ausführen, dennoch besteht bei dieser Methode das Problem, dass vorherige Elemente überschrieben werden, wenn mehrere Elemente nacheinander eingefügt werden. Aufgrund dieses Problems wurde entschieden „Word 2013“ nicht zu unterstützen und stattdessen eine Funktion der Version „Word 2016“ zu verwenden.
Ein weiterer Aspekt sind Unterschiede zwischen einzelnen Versionen hinsichtlich der Performance und Ausführung. So wurden bei Version 2016 im Vergleich zu 2021 längere Ladezeiten und häufig auftretende grafische Fehler festgestellt. Ein Beispiel für einen dieser grafischen Fehler ist, dass bei Version 2016 nach dem Schreiben mehrerer Tabellen nur ein Teil der Tabellen angezeigt wurde. Der Rest der Tabellen wurde dann erst nach der Interaktion mit dem Dokument, beispielsweise durch das Klicken innerhalb des Dokumentes mit der Maus, angezeigt.

4.1.2 Konventionen bei der Erstellung eines Templates

Wie bereits im Abschnitt 3.4 erwähnt, müssen bei der Benutzung von einiger ORKG-Templates Konventionen getroffen werden. Solche Konventionen haben den Nachteil, dass ein Autor eines Dokuments, bei Benutzung eines dieser Templates über die jeweilige Konvention informiert sein muss. Alternativ zu einer Konvention könnte man die benötigten Daten auch mithilfe einer Adressauflösung entnehmen. Beispielsweise kann man den Parameter „Column“ im bereits genannten Beispiel auch durch die Auflösung der angegebenen Adresse, in der Zeile 17 der Codeauflistung 4.1, ermitteln. Auch wenn diese Lösung das Problem einer Konvention umgeht, bringt eine solche Vorgehensweise ein anderes Problem mit sich. Die Qualität der Ergebnisse nimmt in einigen Fällen bei einer Adressauflösungen ab, da die Objekte hinter jeweiligen Adressen nicht an ein jeweiliges Beispiel angepasst sind. So erhält man zum Beispiel bei der Auflösung der Adresse in Zeile 17 das Ergebnis „ORKG synonyms“ anstatt von „ORKG“.

[2]https://learn.microsoft.com/en-us/javascript/api/requirement-sets?view=common-js-preview (Abgerufen: 10.03.2024)

```
...
"P1004": [
        {"@id": "https://orkg.org/resource/R125989"}
    ],
    "P4077": [...],
    "P32": [
        {"@id": "https://orkg.org/resource/R573970"}
    ],
    "PWC_HAS_BENCHMARK": [
        {
            "@id": "_:n2",
            "label": "",
            "@type": [
                "https://orkg.org/class/C14022"
            ],
            "P2005": [
                {"@id": "https://orkg.org/resource/R657829" }
            ],
...
```

Codeauflistung 4.1: JSON Leaderboard Beispiel Ausschnitt Fortsetzung.

4.1.3 Aufwendige Erstellung von Templates

Ein weiteres Problem ist das Erstellen neuerer OOXML-Templates. Das Benutzen von Templates hatte insbesondere den Vorteil der Versionskompatibilität, dennoch ist das Schreiben von Templates mit einem großen Aufwand verbunden. Die OOXML-Templates sind in ihrem Umfang viel größer als einfache API-Aufrufe. Dies wird schnell bei einem Vergleich der Länge beider Codes ersichtlich. Während der in Abbildung 4.3 dargestellter Code, für eine einfache Funktion die mithilfe der Word-API eine Tabelle generiert, in etwa 264 Zeichen auf 8 Zeilen umfasst, beträgt die Länge der .rels Datei des OOXML-Templates allein bereits 685 Zeichen auf 9 Zeilen. Insgesamt umfasst das Template in ungekürzter Fassung sogar circa 1130 Zeilen.
Auch bei einer intensiveren Nutzung von API-Aufrufen, lässt sich dennoch zurzeit selbst mit Word 2021 nicht vollständig auf ein OOXML-Template verzichten, da eine API Funktion zur Erstellung von Graphen derzeit noch nicht unterstützt wird.

4.2 Stärken

Trotz der vielen Schwierigkeiten in der Entwicklung zeigt die Implementierung einige Stärken. Im Folgenden werden die erkannten Vorteile nun präsentiert.

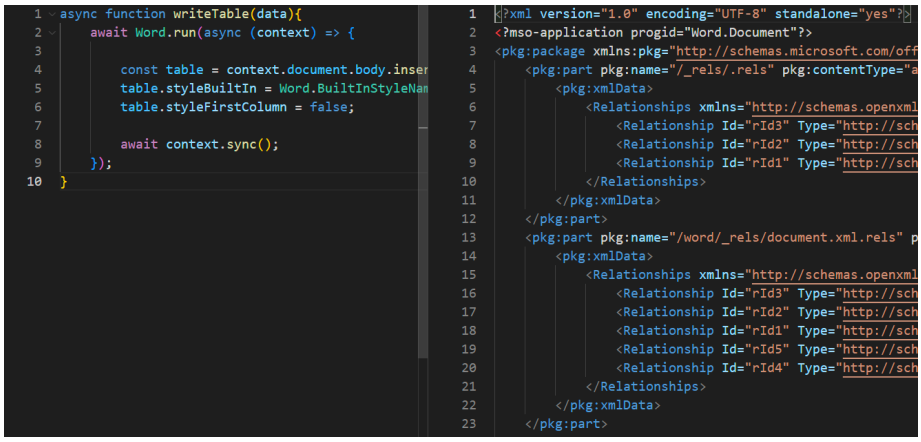


Abbildung 4.3: Vergleich von einer Funktion mit API-Aufruf zu Tabellen Generierung (links) und erste paar Zeilen des verwendeten Tabellen-Template (rechts) bezüglich ihrer Länge.

4.2.1 Weite Versionskompatibilität

Die Unterstützung einer möglichst großen Anzahl an Versionen war für die Entwicklung des Systems eine wichtige Eigenschaft. Trotz den Problemen ist es gelungen das Add-In unter drei von vier möglichen lokalen Versionen zu unterstützen. Da für gewöhnlich ältere API Sets auch auf neueren Versionen unterstützt werden, ist davon auszugehen, dass auch zukünftige Versionen von Word kompatibel sein werden. Neben den genannten Versionen wird des Weiteren von Microsoft angegeben, dass die „Microsoft 365“-, Web-, iPad- und Mac-Versionen ebenfalls die benötigten API's unterstützen. Demnach besteht eine hohe Wahrscheinlichkeit, dass das Add-In auf diesen Versionen ebenfalls funktioniert. Aufgrund von fehlenden Tests lässt sich dies dennoch nicht bestätigen. Eine weite Versionskompatibilität wird als besonders Vorteilhaft angesehen, da ältere Versionen von Microsoft Word noch weiterhin von vielen Nutzern verwendet werden. Die in Abschnitt 4.1.1 genannten Anzeigefehler sind zwar störend, ändern aber nichts an dem funktionalen Umfang der Anwendung und werden demnach nicht als gravierend angesehen.

4.2.2 Wiederverwendbarer Code

Der Programmcode der React-App ist ein weiterer Vorteil, da dieser bei Erweiterungen durch seine stark generische Vorgehensweise und seiner systematischen Unterteilung weitestgehend unverändert bleiben kann. Um diese Eigenschaft weiter zu

erläutern betrachtet man nun verschiedene Szenarien:

- Weitere Template Unterstützung: Für den Fall, dass ein neues Template unterstützt werden soll, muss lediglich eine neue YAML-Spezifikation geschrieben werden. Sollten kein neuer Visualisierungstyp unterstützt werden oder sonstige neue Funktionen, müssen an dem Programmcode keine Änderung vorgenommen werden.

- Neue Funktionen: Wenn ein anderes Datenformat benötigt wird oder weitere Berechnungen ausgeführt werden, muss eine neue Handlerfunktion geschrieben werden und im dritten Codeabschnitt (Abschnitt 3.4) integriert werden. Eine solche Änderung ist mit etwas mehr Aufwand verbunden, muss jedoch nur selten vorgenommen werden.

- Neuer Visualisierungsformat: Sollte ein neues Format für eine Visualisierung hinzugefügt werden, wie zum Beispiel ein neuer Diagrammtyp, dann muss ein neues Template erstellt werden und ein jeweiliger Eintrag in der „getTemplate“ Funktion hinterlegt werden. Falls benötigt müssen auch neue Parameter oder Hilfsfunktionen für die Template-Generierung deklariert werden. Abhängig der Visualisierung kann das Erstellen eines Templates, wie bereits in Abschnitt 4.1.3 erwähnt, sehr mühsam sein, dennoch müsste dies ebenfalls nur selten vorgenommen werden.

- Plattformwechsel: Ein Wechsel der Plattform ist theoretisch auch möglich, da die gesamte Anwendung eine React Web-App ist. Mit ein wenig Aufwand lässt sich das Add-In vollständig von Microsoft Word lösen und stellt somit eine eigenständige webbasierte Anwendung dar. In der Praxis hängt ein solcher Plattformwechsel natürlich stark von der Zielplattform ab. So wäre ein Wechsel auf Latex beispielsweise recht einfach, da es möglich ist ein neues Template zu erstellen, welches Latex Code einer jeweiligen Visualisierung generiert. Als Beispiel nimmt man das Use Case aus Thießen et al. (2023) . Wenn man das Tabellen Template mit dem in Codeauflistung 4.2 dargestellten Template ersetzt und anschließend den generierten XML-Code für die Referenzdaten auf der Konsole ausgibt, erhält man einen Latex Code für eine Tabelle. Dieser generierte Code wurde als Tabelle 4.1 unverändert, mit Ausnahme des Untertitels, in diese schriftliche Arbeit eingefügt. Für das Template musste zusätzlich eine weitere Hilfsfunktion in Typescript definiert werden, diese ist in Codeauflistung 4.3 abgebildet. Auf der anderen Seite gibt es dennoch auch Plattformen, auf denen ein Plattformwechsel deutlich aufwendiger ist. Ein Beispiel dafür ist die Textverarbeitungssoftware Google Docs. Da Google Docs Add-ons als

Apps Script-Projekte erstellt werden. Diese basieren zwar auf JavaScript, dennoch ist unklar wie viel des Codes, insbesondere der Template Generierung, übernommen werden kann[3]. Eine Erstellung von grafischen Objekten sollte aufgrund der XML-Struktur der Dokumente[4] möglich sein, dennoch ist unklar wie Templates in der Apps Script Umgebung gespeichert werden können.

```
\begin{table}
\centering
\begin{tabular}{ {{#each (range 0 (add columnCount 1))}}|c{{/each}}| }
 \hline
 {{#each (getData) as |row|}}
   {{ @root.insertRowLatex row }}
   \hline
 {{/each}}
\end{tabular}
\end{table}
```

Codeauflistung 4.2: Beispiel für ein Table-Template für Latex.

```
const insertRowLatex = (row) => {
    let ret = ''
    const last_index = row.length -1
    for(let i = 0; i<last_index; i++){
        ret += row[i] + ' & '
    }
    return new handlebars.SafeString(ret + row[last_index] + ' \\\\')
}
```

Codeauflistung 4.3: Typescript Code für die Funktion insertRowLatex.

4.2.3 Einfache Benutzung und Anpassungsfähigkeit

Ein weiter Aspekt ist die leicht verwendbare Benutzeroberfläche und die Anpassungsfähigkeit von generierten Objekten. Da nur ein Knopf mit einer passenden Beschriftung auf der Benutzeroberfläche vorhanden ist, ist es klar wie die Anwendung benutzt werden muss. Die Installation sollte aufgrund der Entkopplung des Programmcodes ebenfalls recht einfach sein. Der Benutzer muss zur Bedienung der Anwendung keine Programmierkenntnisse haben und muss lediglich die Manifest Datei in Microsoft Word angeben. Die anpassungsfähigen Visualisierungen sind auch

[3] https://developers.google.com/apps-script/guides/import-export?hl=de (Abgerufen: 10.03.2024)

[4] https://developers.google.com/apps-script/advanced/docs?hl=de (Abgerufen: 10.03.2024)

Tabelle 4.1: Generierte Latex Tabelle aus dem Table-Template. (Nur dieser Untertitel wurde hinzugefügt).

null	ORKG	CS-KG	SemEval-17	SciERC
1	0.618	0.789	0.573	0.668
2	0.604	0.784	0.572	0.651
3	0.611	0.765	0.568	0.65
4	0.597	0.752	0.559	0.642
5	0.605	0.803	0.565	0.644
6	0.597	0.874	0.562	0.663

vom großen Vorteil, da dieses es dem Nutzer erlauben, Visualisierungen an deren Vorstellungen anzupassen. Es lassen sich zudem auch die unpassenden generierten Bezeichnungen (Vergleiche Abschnitt 3.8) ohne großen Aufwand manuell ändern.

4.3 Schwächen

Neben den positiven Eigenschaften der Implementierung wurden dennoch auch einige Schwächen identifiziert, welche nun im weiteren beschrieben werden.

4.3.1 Schreiben von Spezifikationen

Das Schreiben von Spezifikationen ist derzeit ein großer Nachteil, da für jedes einzelne ORKG-Template eine neue YAML-Spezifikation verfasst werden muss. Zudem ist es unklar, wann und wie das Bearbeiten von diesen Spezifikationen stattfinden kann. Auch wenn das Lesen von YAML-Code für gewöhnlich im Vergleich zu anderen Datenformate für Menschen als leichter empfunden wird [5], ist es eine recht unangemessene Vorgehensweise diese Aufgabe den Nutzern zu vergeben. Stattdessen wäre es besser eine automatische Generierung dieser Spezifikationen zu entwerfen. Da derzeit eine vollständige Generierung ohne menschlichen Input nur bedingt möglich ist, sollte ein System entwickelt werden, welche die Generierung stark erleichtert. Eine Idee wäre auf der Website vom ORKG das Erstellen von Visualisierungstemplates zu ermöglichen, in welchen ein Nutzer für ein bestehendes Template die entnehmbaren Daten markieren kann und den Visualisierungstyp angeben kann. Im Hintergrund würde das System dann eine jeweilige YAML-Datei erstellen. Diese Idee lässt sich

[5] https://yaml.org/spec/1.2.2/ (Abgerufen: 10.03.2024)

noch weiter vertiefen indem beispielsweise für einzelne Attribute eigene Spezifikationen definiert werden um das Erstellen neuerer YAML Dateien einfacher und effizienter zu machen. Wie in Abbildung 4.4 dargestellt, kann man beispielsweise in der YAML-Datei vom „Student's t-test“ definieren, dass die Datenwerte aus einem Objekt der Klasse table zu entnehmen sind und dessen Format in einer separaten Datei „table.yaml“ spezifiziert ist. Das gleiche lässt sich dann auch mit dem P-Wert machen indem in der Datei „pValue.yaml“ zusätzliche Daten definiert werden.

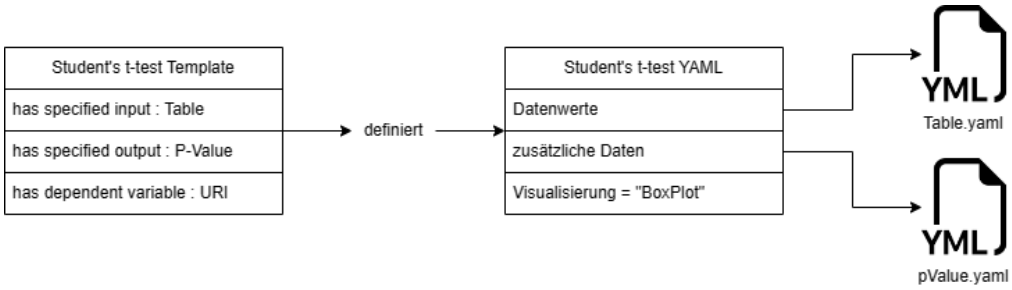


Abbildung 4.4: Möglichkeit zum Vereinfachen von YAML-Spezifikationen.

4.3.2 Konventionen und fehlende Daten

Ein Weiterer negativer Aspekt sind die erwähnten Konventionen (Abschnitt 4.1.2) und unpassenden Daten (Vergleiche Abschnitt 3.8). Der Umgang mit Konventionen lässt sich erleichtern, indem ein Nutzer ausreichend über bestehende Konventionen bei der Benutzung von ORKG-Templates informiert wird. Dennoch wird bei einem solchen Vorgehen nicht garantiert, dass die Nutzer diese Hinweise wahrnehmen oder beachten. Eine andere Möglichkeit wäre das Erzwingen eines bestimmten Formats bei Daten, für welche Konventionen vorgesehen sind und dessen Überprüfung bei Änderungen. Dies verlangt dennoch, dass das jeweilige Format dem System logisch, beispielsweise in Form von regulären Ausdrücke (Sakarovitch, 2006), bekannt ist. Dem entsprechend müssten Nutzer diese Konventionen dann auch in Form eines Schemas definieren.
Im Falle von fehlerhaften Daten kann man insbesondere zwei schwierige Fälle von Daten identifizieren:

1. Nicht vorhandene Daten: Bestimmte Daten, welche in einem Template nicht angegeben werden.

2. Daten ohne Klartextrepräsentation: Daten, welche nur durch das Auflösen einer IRI erhalten werden können.

Ein Beispiel für den ersten Fall ist die Spalten-Oberbezeichnung „dataset“ des Use Cases Thießen et al. (2023) , da diese Art von Information nicht direkt in den JSON-LD Dateien definiert ist. Aufgrund dessen, dass Templates zur Visualisierung möglichst generisch bleiben sollen, ist es zudem unpassend diese Informationen zu entfernen. Eine Option wäre dennoch im Template zu spezifizieren, dass bei undefinierten Werten einzelne Bereiche nicht generiert werden sollen. Der Nachteil dieser Idee ist dennoch, dass die Erstellung von Templates somit stark verkompliziert wird. Der zweite Fall von Daten lässt sich beispielsweise im Use Case Haddad et al. (2016) finden, als Bezeichnung für die Y-Achse des Diagramms. Möglichkeiten mit dem Umgang dieser Daten, ist das definieren einer Konvention oder das auflösen der IRI. Wie aber bereits beschrieben, haben beide Optionen ihre Vor- und Nachteile.

4.3.3 Templates und Funktionserweiterungen

Wie bereits in Abschnitt 4.2.2 beschrieben, kann das Erstellen von neuen Templates und Funktionen sehr aufwendig sein. Aufgrund dessen, dass das derzeitige System nur für wenige Templates ausgelegt ist, müssen eine Vielzahl an weiteren Funktionen und Templates implementiert werden, um einen ausreichend großen Funktionsumfang zu gewährleisten. Das Erstellen von weiterer Templates ist von großer Bedeutung um einen möglichst großen Nutzen für Benutzer erzielen zu können. Die aufwendige Erstellung von Templates (Vergleiche Abschnitt 4.1.3) ist ein weiterer Aspekt, welches die Bereitstellung der Anwendung erschweren kann. Neben dem Hinzufügen weiterer Templates gehört auch das Bearbeiten von bereits vorhanden Templates, falls neue Daten in einem Objekt visualisiert werden müssen.

4.4 Aufgriff der Forschungsfrage

Wie der Abschnitt 3.5 bereits gezeigt hatte, lassen sich mit dem Add-In ORKG-Daten visualisieren. Die visualisierten Daten werden dabei fehlerfrei angezeigt und werden abhängig vom gewählten ORKG-Template dargestellt. Autoren wissenschaftlicher Publikationen sind somit in der Lage die während der Forschung gesammelten Ergebnisse mithilfe eines ORKG-Templates in ein maschinenlesbares Format zu bringen und diese dann für die automatische Visualisierung zu nutzen. Auf dieser Weise müssen Nutzer nicht eigenständig Diagramme in externen Programmen erstellen und vermindern somit die Gefahr von Tippfehlern oder Darstellungsfehler.

Zudem können weitere Vorteile (Abschnitt 4.2) der gewählten Implementation die Verwendung zusätzlich erleichtern, wie die Anpassungsfähigkeit der generierten Abbildungen.

Kapitel 5

Fazit und Ausblick

In diesem Abschnitt wird ein Fazit aus den erzielten Ergebnissen und Erfahrungen gezogen und hinsichtlich der anfangs gestellten Fragen reflektiert (Abschnitt 5.1). Anschließend werden in Abschnitt 5.2 mögliche weiterführende Ansätze genannt, um das entwickelte System zu erweitern, beziehungsweise zu verbessern.

5.1 Fazit

Im Rahmen dieser Arbeit wurden Möglichkeiten untersucht, mithilfe von strukturiertem Wissen und einer Software, Autoren bei der Manuskriptherstellung zu unterstützen. Da es ein Ziel dieser Arbeit ist, die Beteiligung an der Erstellung neuer Beiträge für den ORKG zu steigern, sollte die Anwendung stark auf weitere vom ORKG angebotene Technologien aufbauen. Dazu wurde analysiert inwiefern es möglich ist strukturierte wissenschaftliche Daten mithilfe des ORKG's zu generieren und diese als Eingabe für die zu entwickelnde Software zu nutzen. Es wurde detailliert beschrieben, welche Anforderungen eine solche Anwendung leisten muss und auf welcher Weise und mit welchen Technologien dieses umgesetzt werden kann. Anschließend wurde ein Beispielsystem in Form eines Microsoft Word Add-Ins vorgestellt, welches als Beitrag für diese Arbeit entwickelt wurde. Dafür wurde die Unterteilung des Programmcodes hinsichtlich ihrer Funktion beschrieben und mithilfe von verwendetet Templates und Spezifikationen veranschaulicht. Nach dem Beschreiben der Implementierung wurde zudem eine Beispielsausführung erläutert und mit den erwarteten Resultaten, aus bereits veröffentlichten wissenschaftlichen Arbeiten verglichen. Zum Abschluss wurden Probleme, welche während des Entwicklungsprozesses aufgetreten sind, geschildert und mögliche alternativen Vorgehensweisen genannt. Zudem wurde auch eine Vielzahl an Stärken und Schwächen der Implementierung

genannt. Für die negativen Eigenschaften wurden zusätzlich Lösungsvorschläge in Betracht gezogen.
Die Frage, ob ein solches beschriebenes System zusammen mit den weiteren Angeboten des ORKG mehr Forscher dazu anregt weitere Beiträge dem Wissensgraphen hinzuzufügen, lässt sich zu einem jetzigen Stand nicht bestätigen. Auch wenn sich keine genaue Aussage treffe lässt, ist es möglich das entwickelte Microsoft Word Add-In hinsichtlich vieler Eigenschaften zu bewerten und somit dessen Nutzen abzuschätzen. Wie im Abschnitt 4.2 beschrieben gibt es einige große Stärken an der Implementierung. Die leichte Benutzbarkeit und Verfügbarkeit des Add-Ins sind zwei große Vorteile, welche die Anwendung auf dieser Weise einer möglichst großen Gruppe von Anwendern zur Verfügung stellt. Ein weit verbreiteter Ansatz Visualisierungen für gesammelte Daten zu generieren, ist es ein Programm oder eine Programmierumgebung (Beispielsweise Python) zu Nutzen, um bei manueller Eingabe der Werte eine Visualisierung zu erhalten. Üblicherweise sind solche generierten Dateien von Standard Grafik-Dateiformate wie zum Beispiel PNG, JPEG und SVG. Der große Vorteil an der Visualisierung des implementierten Add-Ins ist dennoch die vollständig anpassbaren generieten Darstellungen. Somit wird für einen Nutzer des Add-Ins beim spezifizieren von Daten und dem Anpassen von Grafiken Zeit gespart.
Bevor die beschriebene Software jedoch vollständig zum Einsatz kommen kann, muss diese noch an einigen Stellen weiterentwickelt werden. Zuallererst muss überlegt werden wie mit den YAML-Spezifikationen umgegangen wird, insbesondere muss dabei entschieden werden wie neue Spezifikationen von Templates möglichst effizient definiert werden können und wo diese gespeichert werden. Im Weiteren müssen zudem eine Vielzahl weiterer Templates und verschiedene Visualisierung-Typen implementiert werden, damit es für einen Großteil der Templates eine geeignete Visualisierung gibt. Auch bei bereits umgesetzten Templates müsste überlegt werden, ob zusätzlich weitere Funktionen in der Darstellung angezeigt werden sollen.
Zusammenfassend kann man deuten, dass die implementierte Software zur Unterstützung bei der Manuskripterstellung mithilfe von wissenschaftlichem Wissen für Nutzer hilfreich sein kann und somit ein möglicher Anreiz zur Nutzung der ORKG-Dienste ist. Trotzdem müssen für einen Erfolg dieser Anwendung noch weitere Erweiterungen vorgenommen werden.

5.2 Ausblick

Wie bereits im vorherigen Abschnitt erklärt ist es für den Erfolg der Implementierung wichtig, dass der Umfang der Anwendung erweitert wird und eine Lösung für den Umgang mit den YAML-Spezifikationen gefunden wird. Eine Idee dafür wur-

de bereits in Abschnitt 4.3.1 vorgestellt. Neben einer Lösung für die Spezifikationen muss auch der Umgang mit den Konventionen betrachtet werden. Dafür wurde im Abschnitt 4.1.2 bereits ein Vorschlag genannt. Des Weiteren müssen auch weitere Templates für die Visualisierung erstellt werden, damit zusätzliche Darstellungen unterstützt werden können, wie zum Beispiel Histogramme, Netzdiagramme oder möglicherweise sogar spezielle Ablaufdiagramme oder Kartendiagramme. Eine Unterstützung von möglichst vielen Darstellungstypen ist besonders wichtig, damit möglichst jedes ORKG-Template durch eine passende Visualisierung dargestellt werden kann. Zusätzlich müssen Tempaltes aber auch viele Sonderfälle unterstützen, um Nutzern möglichst viel Aufwand bei der Anpassung abzunehmen.
Neben der Erweiterung von bereits vorhandenen Funktionen, können auch noch weitere Eigenschaften implementiert werden, um die Ergebnisse zu verbessern. Viele dieser Ideen wurden bereits im Entwicklungsprozess in Erwägung gezogen. Zum Beispiel wurde überlegt inwiefern LLM's als Werkzeug genutzt werden können um beispielsweise Beschreibungen von Diagrammen zu generieren. Dazu wurden verschiedene Eingaben in ChatGPT 3.5[1] für das Use Case aus Haddad et al. (2016) ausprobiert. Die Qualität der Ergebnisse waren dennoch nicht überzeugend, insbesondere beim Generieren von einer Beschreibung in deutscher Sprache. Mit der Veröffentlichung von GPT-4[2] und weiteren LLM, sowie dem Potential einer solchen Erweiterung, sind weitere Tests in diesem Bereich dennoch von großem Wert.
Ein weiteres Beispiel ist die Veröffentlichung der Anwendung auf weiteren Plattformen, wie bereits im Abschnitt 4.2.2 erwähnt. Das unterstützen von weiteren Plattformen ist eine wichtige Eigenschaft, welche gewährleistet, dass Nutzer verschiedener Plattformen von dem Angebot profitieren können.

[1] https://chat.openai.com/ (Abgerufen: 10.03.2024)
[2] https://openai.com/gpt-4 (Abgerufen: 10.03.2024)

Literatur

Anfuso, M. (2023). Born-reusable scientific knowledge: Concept, implementation, and applications. https://doi.org/10.15488/14222

Auer, S., Oelen, A., Haris, M., Stocker, M., D'Souza, J., Farfar, K. E., Vogt, L., Prinz, M., Wiens, V., & Jaradeh, M. Y. (2020). Improving Access to Scientific Literature with Knowledge Graphs. https://doi.org/doi:10.1515/bfp-2020-2042

Bless, C. (2022). SciKGTeX - A LATEX Package to Semantically Annotate Contributions in Scientific Publications. https://doi.org/10.15488/12462

Boubakri, Z. (2022). The ORKG R Package and Its Use in Data Science. https://doi.org/10.15488/13072

Boyer, J. (2001, März). *Canonical XML Version 1.0* (Techn. Ber.) (W3C Recommendation). World Wide Web Consortium (W3C). https://www.w3.org/TR/2001/REC-xml-c14n-20010315

Boyer, J., 3rd, D. E. E., & Reagle, J. (2002, Juli). *Exclusive XML Canonicalization Version 1.0* (Techn. Ber.) (W3C Recommendation). World Wide Web Consortium (W3C). https://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/#sec-Terminology

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008, November). *Extensible Markup Language (XML) 1.0 (Fifth Edition)* (Techn. Ber.) (W3C Recommendation). World Wide Web Consortium (W3C). https://www.w3.org/TR/REC-xml/

ECMA International, T. (2017). ECMA-404 The JSON data interchange syntax. *2*. https://ecma-international.org/publications-and-standards/standards/ecma-404/

ECMA international, T. (2021). *ECMA-376 Office Open XML file formats*, *5*. https://ecma-international.org/publications-and-standards/standards/ecma-376/

Fathalla, S., Vahdati, S., Auer, S., & Lange, C. (2017). Towards a Knowledge Graph Representing Research Findings by Semantifying Survey Articles. In J. Kamps, G. Tsakonas, Y. Manolopoulos, L. Iliadis & I. Karydis (Hrsg.), *Research and Advanced Technology for Digital Libraries* (S. 315–327). Springer International Publishing.

Haddad, S., Wang, Y., Galy, B., Korf-Klingebiel, M., Hirsch, V., Baru, A. M., Rostami, F., Reboll, M. R., Heineke, J., Flögel, U., Groos, S., Renner, A., Toischer, K., Zimmermann, F., Engeli, S., Jordan, J., Bauersachs, J., Hentze, M. W., Wollert, K. C., & Kempf, T. (2016). Iron-regulatory proteins secure iron availability in cardiomyocytes to prevent heart failure. *European Heart Journal*, *38*(5), 362–372. https://doi.org/10.1093/eurheartj/ehw333

ISO, I. 4. 9. (2022). *ISO 26324:2022 Information and documentation Digital object identifier system*, *2*. https://www.iso.org/standard/81599.html

Peterson, W. (2001). *Almost Perfect: How a Bunch of Regular Guys Built Word Perfect Corporation*. DIANE Publishing Company. https://books.google.de/books?id=CYqjngEACAAJ

Sakarovitch, J. (2006). The Language, the Expression, and the (Small) Automaton. In J. Farré, I. Litovsky & S. Schmitz (Hrsg.), *Implementation and Application of Automata* (S. 15–30). Springer Berlin Heidelberg.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Champin, P.-A., & Lindström, N. (2020, Juli). *JSON-LD 1.1 A JSON-based Serialization for Linked Data* (G. Kellogg, P.-A. Champin & D. Longley, Hrsg.; Techn. Ber.) (W3C Recommendation). World Wide Web Consortium (W3C). https://www.w3.org/TR/json-ld/

Thießen, F., D'Souza, J., & Stocker, M. (2023). Probing Large Language Models for Scientific Synonyms. *SEMANTICS Workshops*. https://api.semanticscholar.org/CorpusID:265068593

Tsang, C. D. (2000). Microsoft first generation: The success secrets of the visionaries who launched a technology empire. J. Wiley & Sons. Verfügbar 20. Februar 2024 unter https://archive.org/details/microsoftfirstge00cher/page/37/mode/2up

Unwin, A., & Kleinman, K. (2021). The iris data set: In search of the source of virginica. *Significance*, *18*. https://api.semanticscholar.org/CorpusID:244763032

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., Bouwman, J., Brookes, A. J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C. T., Finkers, R., … Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, *3*(1), 160018. https://doi.org/10.1038/sdata.2016.18