Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Konzeption einer Ontologie zur Beschreibung von Daten der CAE Domäne

Denis Moslavac

Studiengang:	Informatik
Prüfer/in:	Herr PD Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Julian Ziegler, M.Sc.
Beginn am:	12. August 2020
Beendet am:	12. Februar 2021

Kurzfassung

In der heutigen Zeit haben Menschen viel mehr Zugriff auf Daten innerhalb eines Tages, als früher die meisten in ihrer gesamten Lebenszeit. Problematisch hierbei ist, dass diese Daten in verschiedensten Formen und Formaten gespeichert sind und somit das Verständnis der Beziehung zwischen diesen Daten sehr erschwert wird. Handelt es sich bei den Daten nun um komplizierte industrielle Daten (CAE-Daten), so wird das Verständnis noch problematischer.
Im Rahmen dieser Bachelorarbeit wird ein Konzept und ein Prototyp einer Ontologie zur Beschreibung von Daten der CAE-Domäne entwickelt. Eine Ontologie ist eine formale Repräsentation von Wissen innerhalb einer Domäne, welches entweder bestimmte Eigenschaften einer Domäne oder die Domäne an sich beschreibt. CAE beschreibt dabei die Technologie, Simulationen aus CAD Modellen durchzuführen und sie zu analysieren, um hierdurch Erkenntnisse über das Produkt unter bestimmten Bedingungen zu erfahren. Die hier entwickelte CAE-Ontologie beschreibt den Zusammenhang der in der Domäne anfallenden Daten. Des Weiteren wird die CAE-Landschaft analysiert, da sie für die Entwicklung der Ontologie ausschlaggebend ist.
Die CAE-Ontologie ist dabei in drei Bereiche eingeteilt. Der Spezifizierung, der Evaluation und dem Tool. Die Klasse der Spezifizierung soll dabei den Bereich der Auftraggeber darstellen. Hier werden deren Daten gespeichert, wie die Anforderungen an das Projekt oder den Vertrag. Der Evaluationszweig befasst sich mit dem Speichern von Evaluationsdaten und deren Beziehung zueinander. Alle Beziehungen der Klasse Tool beschäftigen sich mit den Simulationsdaten eines CAE-Projekts. Darunter enthalten sind die CAE- und CAD-Modelle, die Beschreibungen wie sie aufgebaut sind und welche Elemente diese Modelle beeinflussen.
Die Korrektheit der Ontologie wird in der Diskussion mithilfe einer Datenbankabfrage in SPARQL, sowie einer Vorstellung automatisch generierter Inferenzen mit SWRL gezeigt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

CAD Computer-Aided Design. 11, 29

CAE Computer-Aided Engineering. 11, 29

CAM Computer Aided Manifacturing. 11, 32

CAT Computer-Aided Testing. 11

CFD Computational Fluid Dynamics. 5, 11

DMU Digital Mock-Up. 5, 11

DTD Dokumenttyp-Definition. 11, 22

FEM Finite Elements Method. 5, 11

HTML Hypertext Markup Language. 11, 22

JT Jupiter Tesselation. 11, 41

OWL Web Ontology Language. 11, 21

PDM Product Data Manager. 11, 30

RDF Ressource Description Framework. 11, 21

RDFS RDF-Schema. 11, 25

SDM Simulation Data Manager. 11, 30

SGML Standard Generalized Markup Language. 11, 22

SPARQL SPARQL Protocol and RDF Query Language. 11, 58

STEP Standard for the exchange of product model data. 11, 35

SWRL Semantic Web Rule Language. 11, 56

TDM Team Data Manager. 11, 30

URI Uniform Resource Identifier. 11, 24

URL Uniform Resource Locators. 11, 24

W3C World Wide Web Consortium. 11, 23

XHTML Extensible Hypertext Markup Language. 11, 22

XML Extensible Markup Language. 11, 21

1. Einleitung

CAE ist bereits seit vielen Jahren ein wichtiger und unabdingbarer Bestandteil für die moderne Produktentwicklung. So simuliert beispielsweise die Autoindustrie ihre Unfälle und vermeidet Kosten in Millionenhöhe, sowie Zeit bei der Konstruktion der Crashobjekte [al99]. Zwar helfen diese Einsparungen, um Simulationen schneller, effektiver und öfter durchzuführen, jedoch steigt die Komplexität der Daten. In jeder Simulation sind viele Programme involviert. Dies führt zu einer großen Anzahl an heterogenen Daten. Um diese Daten zu strukturieren, werden u.a. Ontologien entwickelt. Die Ontologie, die hier vorgestellt wird, beschreibt den Aufbau der Daten und deren Beziehungen zueinander. Eine Ontologie ähnelt somit einem Metamodell, jedoch mit besonderem Fokus auf die Beziehungen der Daten. Metadaten sind hierbei eine wichtige Voraussetzung zum Verständnis der Daten, deren Exploration und somit auch für die darauffolgende Datenanalyse.
So existieren Ontologien in dieser Domäne, jedoch beziehen sie sich meistens auf den Datenaustausch zwischen einzelnen Systemen. Bei den Systemen handelt es sich beispielsweise um CAD-Entwicklungstools von verschiedenen Firmen. Eine umfangreiche Betrachtung der CAE-Domäne mit allgemeinen Fokus auf dessen Daten, wurde noch nicht observiert. Zusätzlich wurde in Kooperation mit einem Industriepartner an der Universität Stuttgart ein graphbasiertes System entwickelt, um solche CAE-Daten zu verwalten [ZRKM20]. Für dieses graphbasierte Metamodell wird im Rahmen dieser Arbeit ein Konzept einer Ontologie entwickelt, die eine semantische Beschreibung der heterogenen Daten erlaubt.
Zunächst wurde anhand mehrerer Datensätze die CAE-Landschaft charakterisiert. Die daraus gewonnenen Erkenntnisse der Anforderungen wurden abgeleitet, in die Ontologie integriert und anschließend diskutiert. Eingeteilt ist die Arbeit in vier Kapitel.
Kapitel 1 widmet sich der Beschreibung von Ontologien. Es werden grundlegende Konzepte, verschiedene Arten von Ontologien und das Semantic Web betrachtet, das ein wichtiger Anwendungsbereich von Ontologien ist. Da in dieser Arbeit eine Ontologie konstruiert wird, klärt das Kapitel unter anderem auch auf, welche Sprache zur Konstruktion gewählt wurde und aus welchen Grund diese benutzt wird.
Kapitel 2 beschäftigt sich detailliert mit der CAE-Domäne. Teil dieser Arbeit ist die genaue Auseinandersetzung dieser Domäne, um die Ontologie entwickeln zu können. Der Fokus liegt dabei auf den grundlegenden Aufbau CAE integrierter Systeme, deren Simulationen und der dazugehörigen CAE-Modelle.
Kapitel 3 stellt verwandte Arbeiten und deren Einfluss auf diese Arbeit vor.
In Kapitel 4 wird die entwickelte Ontologie vorgestellt und basierend der Anforderungen aus Kapitel 2 und der Datensätze begründet. So wird einerseits die Ontologie in Form eines ER-Modells veranschaulicht und andererseits werden grundlegenden Konzepte der OWL-Datei, die die Ontologie kodiert, mithilfe von Beispielcode erklärt. Zum Schluss folgt eine Diskussion, die die Funktionalität und Korrektheit der Ontologie durch praktische Datenbankabfragen und automatisch generierten Referenzen zeigt.

2. Ontologie

Die Menschheit bemüht sich schon lange damit, Sachverhalte zu kategorisieren. In der Philosophie ist der Begriff „Ontologie“ bereits länger bekannt, und heißt dem Wortlaut nach so viel wie *die Lehre vom Sein*. Zwar wurde der Begriff erst im späten 17 Jahrhundert von Christian Wolf gebräuchlich, jedoch ist die theoretische Beschäftigung der Welterfassung ungefähr 2500 Jahre alt. So beschäftigte sich schon Parmenides (ca. 520–460 v. Chr.), ein griechischer Vorsokratiker, mit der Kategorisierung und Einteilung gewisser Prozesse, Bereiche und deren Relationen zueinander [al14].
Mit zunehmender Digitalisierung und der damit verbundenen Ausweitung der Daten, wird die Notwendigkeit zur Kategorisierung auch in der Informatik deutlich. Während Ontologien in der Philosophie der reinen Wissensrepräsention dienen, sind sie in der Informatik sehr nutzerorientiert und oft für bestimmte Anwendungen entworfen.
Dieses Kapitel besitzt drei Abschnitte. Im ersten Abschnitt werden die grundlegenden Konzepte einer Ontologie erklärt und welche Elemente sie enthält. Der zweite Abschnitt beschreibt verschiedene Möglichkeiten Ontologien zu kategorisieren und erläutert jede davon. Im dritten Abschnitt werden Ontologien im Zusammenhang des Semantic Webs erklärt. Da das Semantic Web ein großes Anwendungsgebiet ist, folgt in Abschnitt 2.3. eine Erläuterung, kombiniert mit einer Auseinandersetzung der Sprache, die für die Kodierung der Ontologie benutzt wird.

2.1. Grundlegende Konzepte

Eine Ontologie ist eine formale Repräsentation von Wissen innerhalb einer Domäne, das entweder bestimmte Eigenschaften einer Domäne oder die Domäne an sich beschreibt. Formal wird das realisiert, indem die Entitäten der Domäne, deren Eigenschaften und deren Beziehung untereinander beschrieben werden. Ontologien sind aufgebaut aus Aussagen, die auch *Tripel* genannt werden. Ein Tripel besteht aus einem Subjekt, einem beschreibenden Prädikat und einem Objekt, das in einer weiteren Aussage wieder als Subjekt dienen kann.
Modellierungen sind wichtige und gängige Beschreibungssprachen für jede übliche Wissensrepräsentation (z.B. ER-Diagramme, Klassendiagramme, Metamodelle, usw.). Eine Ontologie ist dabei ein formalisierter Ausdruck einer Modellierung. So werden mit grundlegenden Bestandteilen ein einheitliches und zusammengreifendes Begriffssystem entwickelt.Darunter gehören:

- **Klassen:** Eine Klasse ist eine abstrakte Beschreibung von Dingen. So definiert eine Klasse eine Eigenschaft von einer Anzahl an Objekten oder fasst diese in eine Klasse zusammen. Wesentliches Ziel der Objektorientierung, ist die schematische Übersichtlichkeit eines Bereichs. Auch können Klassen als sog. Unterklassen dienen und in eine Oberklasse zusammengefasst werden. Vergleichbar ist dies mit den objektorientierten Programmiersprachen wie Java, die ebenfalls eine solche Struktur aufweisen. Eine Ontologie kann somit hierarchisch aufgebaut sein und einer Taxonomie ähneln.

- **Attribute**: Auch hier lassen sich die Attribute im Kontext der Ontologie mit den Attributen in Java vergleichen. Diese spezifizieren die Individuen (Objekte) einer Klasse und geben ihr somit eine bestimmte Eigenschaft.
- **Relationen**: Eine Relation beschreibt die Beziehung zwischen zwei oder mehreren Klassen. Dabei unterscheidet man zwischen einer *hierarchischen* und einer *assoziativen* Relation.
  Eine hierarchische Relation organisiert eine Baumstruktur (Taxonomie). So gibt es, wie bereits erwähnt, Subklassen, die in Superklassen zusammengefasst werden können und auch deren Attribute erben.
  Eine Assoziative Relation setzt Klassen in Beziehung und das über Baumstrukturen hinweg. Es gibt dabei verschiedene Arten von assoziatven Relationen. Häufig lassen sich laut [Ste20] folgende finden:
  - *Nominative Relationen:* Diese beschreiben die Namen der Klasse.
    (z.B. *Person; hatName; Name*)
  - *Lokative Relationen*: Diese beschreiben den örtlichen Bezug einer Klasse in Relation mit einer anderen.
    (z.B. *Haus; hatAnschrift; Adresse*).
  - *Assoziative Relationen*: Diese repräsentieren Funktionen und Prozesse, in den die Klassen involviert sind oder weitere Eigenschaften der Klasse.
    (z.B. *Arbeiter; baut; Haus*
    Haus; schütztVor; Wetter).

  Genauso wie bei den Klassen, können auch Relationen an sich hierarchisch aufgebaut sein. So kann *hatName* unterteilt werden in *hatMaterialName*. Des Weiteren können Relationen, wie Klassen, auch Eigenschaften besitzen, die diese und somit die Beziehung der jeweiligen Klassen genauer beschreiben. Unter anderem lässt sich sagen [Ste20]:
  - Ob eine Relation universell notwendig ist. Zum Beispiel, dass (*Wand; hatMaterialName; MaterialName*) für jede Wand gilt und somit universell ist.
  - Ob eine Relation optional ist. Man möchte zum Beispiel mit (*Wand; hatTapete; Tapete*) beschreiben können, dass Wände die Möglichkeit haben eine Tapete zu besitzen, da nicht jede Wand eine enthält.
  - Ob eine Relation transitiv ist. Das heißt, wenn *b* eine Relation ist und (*a; c; d*) jeweils Klassen sind, ob dann bei (*a; b; c*) und (*c; b; d*) auch *a b d* gilt. Hierarchische Relationen haben dabei immer diese Eigenschaft.
- **Axiome**: Axiome stellen Bedingungen für Klassen, Relationen oder Attributen dar. Durch sie soll ihr Gebrauch näher spezifiziert werden. Diese Regeln sorgen für die Konsistenz einer Ontologie. Beispielsweise lassen sich durch Axiome darstellen, dass eine Zahl nicht gleichzeitig gerade und ungerade sein kann. Aber nicht nur solche Widersprüche werden vermieden. Beziehungen zwischen Klassen sind auch Axiome. So kann auch festgelegt werden, dass beispielsweise ein Arbeiter eine Arbeitsstelle haben muss, um überhaupt ein Arbeiter sein zu können. Gewisse Axiome werden jedoch nicht in eine Ontologie miteinbezogen, da sie für die Allgemeinheit selbstverständlich ist. Beispielsweise kann eine Person nicht an zwei Orten gleichzeitig sein. Diese Annahme ist zwar richtig, aber für die Ontologie unwichtig. [al17].

2.2. Arten von Ontologien

Mit zunehmender Digitalisierung und der damit verbundenen Anzahl an stetig steigenden Daten, gewinnen Ontologien immer mehr an Relevanz. So werden sie mit der Zeit stets häufiger verwendet, wodurch der Drang zur Klassifizierung von Ontologien steigt. Dafür gibt es verschiedene Ansätze zur Ordnung, basierend auf verschiedene Sichtweisen. Hier werden zwei vorgestellt und erklärt:

2.2.1. Lightweight / Heavyweight Ontologien

Ob eine Ontologie *Lightweight* oder *Heavyweight* ist, hängt von dem Grad ihrer Ausdruckskraft und Formalisierung ab. Ontologien, die eine große Anzahl an Axiomen besitzen und somit Wissen modellieren, indem sie die Semantik einer Domain stark beschränken, werden als Heavyweigth bezeichnet.
Ontologien, die wiederum wenig Axiome benutzen, um Wissen zu modellieren und die Bedeutung der Domain zu erklären, werden als Lightweight bezeichnet. Lightweight Ontologien sind in den meisten Fällen hierarchisch aufgebaut mit wenigen logischen Relationen zwischen den einzelnen Klassen. Hinzukommt, dass Heavyweight Ontologien durch den Umfang an Axiomen auch korrespondierend aufwendiger zu konstruieren sind. So sind Heavyweight Ontologien durch ihre große Anzahl an Axiomen zwar korrekter und für den Computer einfacher zu überprüfen, jedoch sind Lightweight Ontologien für die meisten Anwendungsgebiete ausreichend. Davies, Fensel et al. betonten hierbei die Wichtigkeit der Lightweight Ontologien:

> „*We expect the majority of the ontologies on the Semantic Web to be lightweight. […] Our experiences to date in a variety of Semantic Web applications (knowledge management, document retrieval, communities of practice, data integration) all point to lightweight ontologies as the most commonly occurring type*“ [**Meta**].

2.2.2. Top-Level-, Domain-, Task-, Application Ontologien

Der Klassifizierungsansatz für Ontologien anhand des Grades ihrer Abhängigkeit von einem bestimmten Standpunkt oder einer spezifischen Aufgabestellung ist eine weitere Variante, um Ontologien zu ordnen. Abbildung 2.1 beschreibt die Relation der einzelnen Kategorisierungen der Ontologien damit zusammenhängend, wie gegenstandsbezogen, aufgabenbezogen, spezifisch und generisch diese sind. Es wird also differenziert, ob eine Ontologie eher eine Aktivität oder eine sachliche Domäne beschreibt und wie spezifisch sie das macht. Dabei wird unter vier verschiedenen Ontologiearten unterschieden:

- Eine *Top-Level Ontologie* kann als Ontologie mehrerer Ontologien betrachtet werden, die die selbe oder ähnliche Domänen beschreiben. Haben Ontologien ähnliche Domänen, so teilen sie sich auch viele Eigenschaften. Dies können Klassen, Beziehungen oder Axiome sein. Eine Top-Level Ontologie beschreibt und definiert dabei die gemeinsamen Regeln und allgemeine Objekte dieser Ontologien. Das beste Beispiel dafür, sind die Top-Level Ontologien, die alle Ontologien an sich beschreiben. Die RDFS-Ontologie (abrufbar über `https://www.w3.`

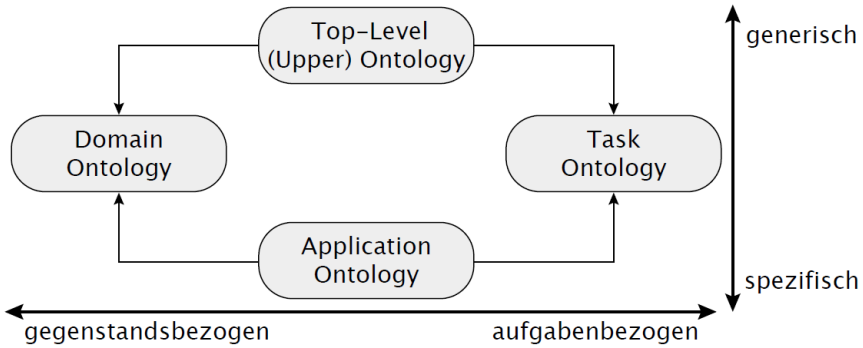

Abbildung 2.1.: Grafische Darstellung von Top-Level-, Domain-, Task-, Application Ontologien (übernommen aus[al17])

org/2000/01/rdf-schema#) definiert beispielsweise was eine Klasse generell ist. Vergleichbar ist eine Top-Level Ontologie mit einem Metametamodell, während Ontologien, die von ihr beschrieben werden, nur Metamodelle sind.

- Eine *Domain Ontologie* und eine *Task Ontologie* besitzen die selbe generische Beschreibungstiefe. Der Unterschied zwischen den beiden ist lediglich das Anwendungsgebiet. Während die Domain Ontologie sehr gegenstandsbezogen ist und sachliche Beziehungen beschreibt, stellt die Task Ontologie die Relationen eines Prozesses dar und ist sehr aufgabenbezogen. Eine Ontologie, die festlegt wie ein Haus aufgebaut ist, wäre ein Beispiel für eine Domain Ontologie, während die Task Ontologie beispielsweise beschreiben würde, welche Arbeitskraft was am Haus bauen würde.
- Die *Application Ontologie* ist sehr spezifisch und beschreibt im genausten einen Ablauf oder eine Domäne. Ein Beispiel wäre, welche genauen Aufgaben der Elektriker am Hausbau hätte.

2.3. Semantic Web

Ein weiterer wichtiger Baustein ist der Anwendungsbereich der Ontologie. Ontologien werden überall dort verwendet, wo es wichtig ist, die Semantik von Daten nutzbar zu machen. Das Semantic Web ist dabei das größte Anwendungsgebiet der Ontologien. Aus diesem Grund muss hierauf ein Schwerpunkt gelegt werden. Die Beschreibungen der verschiedenen Beschreibungssprachen orientieren sich an den Quellen [al14], [KSc07], [al08], [Kar] und [Sic14].
Die Möglichkeit der unmittelbaren Informationsbeschaffung ist für die heutigen Gesellschaftsformen - und Strukturen essentiell und unabdingbar. Nicht nur private Haushalte profitieren von der Möglichkeit der grenzenlosen Informtionsbeschaffung durch das Internet. Auch die Industrie zieht hieraus ihre Vorteile. Das Internet liefert eine große Menge an Daten und Informationen über Daten. Für eine Maschine ist jedoch die Deutung dieser Daten schwierig. So kann ein Mensch den Informationsgehalt eines Dokuments leicht erkennen, während die Maschine nur Zeichenketten vergleicht. Unterschieden wird auch hier zwischen dem *Semantic Web* und dem *Syntactic Web*. Der Unterschied wird anhand eines Beispiels von J. Busse et al. [al14] erklärt:

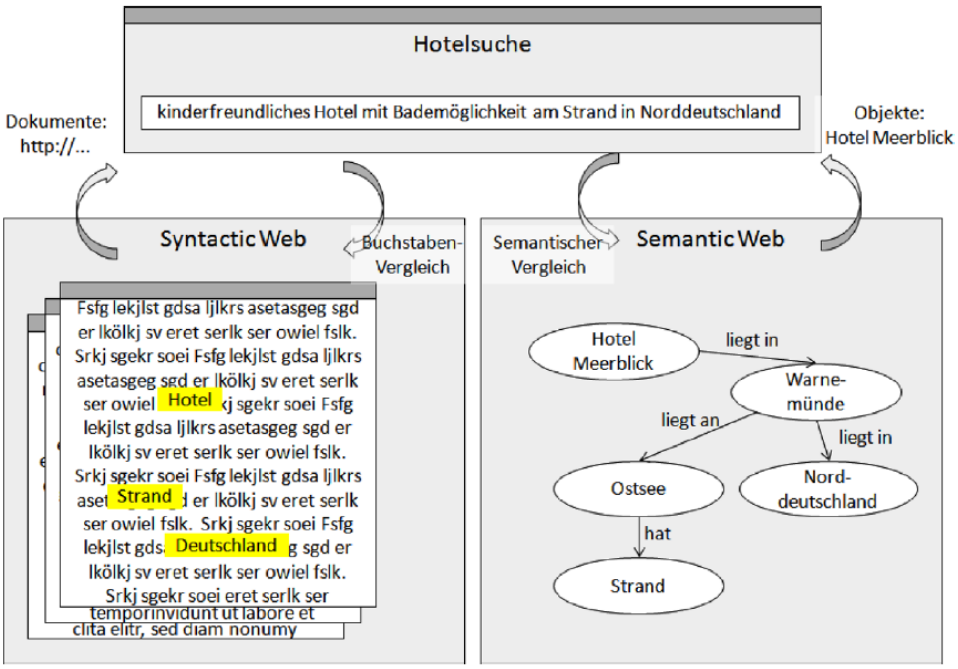

Abbildung 2.2.: Syntactic vs. Semantic Web (übernommen aus [al14])

Abbildung 2.2 verdeutlicht den Unterschied zwischen dem Syntactic Web und dem Semantic Web. Hierbei handelt es sich um eine Hotelsuche, indem eine Person nach einem „kinderfreundlichen Hotel mit Bademöglichkeit am Strand in Norddeutschland“ sucht.
Das Syntactic Web vergleicht hierbei nur Zeichenketten. Dem Nutzer kann hierbei ein Dokument in Form einer Rezension für ein Hotel in Dubai angezeigt werden. Ihm wird dieses Ergebnis jedoch nur deshalb angezeigt, weil die Rezension eine Mehrzahl der vom Nutzer erfragten Wörter enthält. Beispielsweise wurden in der Rezension die Wörter „Bademöglichkeit am Strand“, „Kinder“ und „Deutschland“ gefunden. Bei genauerer Betrachtung stellt man jedoch fest, dass der Autor der Rezension aus Deutschland kommt und angegeben hat, ohne Kinder gereist zu sein. Hier offenbart sich ein lediglich schlichter Vergleich mit einfachen Zeichenketten, welcher jedoch von der eigentlichen Suche nach einem komplexeren Sinngehalt lösgelöst ist.
Wird der gleiche Sinngehalt in einem Reisebüro erfragt, wird der Kundenbetreuer den Sinn hinter der Anfrage verstehen und die richtigen Ergebnisse liefern. Eine Semantic Web-Anwendung für Hotelsuche versucht auch hierbei mithilfe von Ontologien den Kontext der Anfrage nachzuvollziehen. Eine Anwendung könnte dabei die Sprache analysieren (Adjektive, Adverbiale Bestimmungen, usw.), Synonyme finden und ersetzen, durch Referenzen in der Ontologie schlussfolgern und anschließend die gefundenen Ergebnisse ableichen [al14].
Die nachfolgenden Abschnitte liefern einen Einblick in die Beschreibungssprachen Extensible Markup Language (XML), Ressource Description Framework (RDF) und Web Ontology Language (OWL). Unter anderem wird auch geklärt, wieso welche Beschreibungssprachen für die hier entwickelte Ontologie unzureichend sind.

2.3.1. XML

*XML*ist eine Markup- und Meta-Sprache, die dafür zuständig ist, bestimmte Teile von Texten mit zusätzlichen Informationen zu versehen. Dies wird mit sog. Tags erreicht (wie in *Hypertext Markup Language (HTML)*). Dadurch können Texte hierarchisch strukturiert werden und für eine gute Lesbarkeit für Mensch und Computer sorgen. XML basiert auf *Standard Generalized Markup Language (SGML)* und ist eine vereinfachte und verkürzte Form dessen. So ist auch jedes XML-Dokument gleichzeitg ein SGML-Dokument. Wie im Namen bereits erwähnt, ist XML extensible, also erweiterbar. So kann man mit XML die Markup Sprache HTML definieren (*Extensible Hypertext Markup Language (XHTML)*). Man bezeichnet XML auch deshalb als Meta-Sprache, die zur Erstellung von Markup-Sprachen (wie HTML) dient [al08].

Eine *Deklaration* ist in den meisten Fällen der Anfang einer XML-Datei. Mit den Flags *version* und *encoding* erhält sie die Information, um welche XML-Version es sich handelt und wie sie kodiert ist. Das letzte Flag *standalone* kann auf „yes“ oder „no“ gesetzt werden. Ist dieser auf „no“ gesetzt, so heißt dies, dass eine externe Dokumenttyp-Definition (DTD) verwendet werden muss. XML ist nämlich sehr flexibel. Viele XML-Anwendungen enthalten das dagegen nicht, oder nicht in dem Maße. Ein *DTD* ist dabei eine formale Syntax, in der geschrieben steht, wie und wo Elemente auftauchen und welchen Inhalt sie haben dürfen. Sie ist optional und kann weggelassen werden und wird dann standardmäßig auf „no“ gesetzt.
Elemente sind Bausteine von XML-Dokumenten und stukturieren den Text. Sie starten überlicherweise mit einem Start-Tag (<name>) und enden mit einem End-Tag (</name>). Die Zeichen zwischen den Tags bilden den Inhalt. Ein Element kann aber auch inhaltslos sein, wobei man das Element dann abkürzen kann (<name />). Elemente sind case-sensitive, müssen mit einem Buchstaben oder einem Unterstrich anfangen, können nicht mit der Zeichenkette „xml“ beginnen und beinhalten keine Leerzeichen.
Elemente können hierbei auch mit Zusatzinformationen versehen werden. Dies wird mit den *Attributen* umgesetzt. Attribute müssen immer in Anführungszeichen gesetzt werden [al08].
Abbildung 2.3 zeigt ein kleines Beispiel für ein solches XML Dokument.

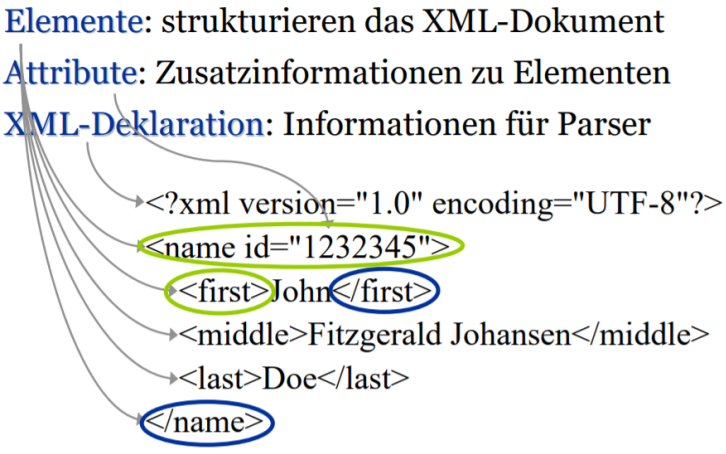

Abbildung 2.3.: Ein Beispiel für ein XML Dokument (übernommen aus [KSc07]).

XML ist laut P.H. et al. eine sehr intuitive und einfache Sprache, wird aber nicht für Ontologien im Semantic Web verwendet.
XML bietet die Grundbausteine, ist jedoch aus der Sicht des Semantic Webs nicht besser als die natürliche Sprache. Die Tags haben zwar für die Menschen eine Bedeutung, für die Maschine jedoch immer dieselbe Struktur. Mit XML ist es beispielsweise nicht möglich automatische Referenzen zu bilden, die jedoch für eine Ontologie wichtig sind. Aus diesem Grund bildet XML lediglich die Grundlage, um weitere Sprachen zu definieren [al08].

2.3.2. Ressource Description Framework

RDF ist ein vom World Wide Web Consortium (W3C) entwickeltes Framework, um Metadaten im Semantic Web zu beschreiben. Während XML Informationen in Baumstrukturen darstellt, beschreibt ein RDF-Dokument einen gerichteten Graphen mit Knoten, Kanten und eindeutigen Bezeichnern. Dies ermöglicht den Austausch, die Wiederbenutzung und die Kodierung von Metadaten. Dabei wird jedoch XML für das Prozessieren und den Austausch von Metadaten als Syntaxgrundlage benutzt.
RDF wurde für die Beschreibung von allgemeinen Beziehungen zwischen den Ressourcen entwickelt. Die hierarchische Baumtruktur von XML ist dabei problematisch, da im Internet Informationen oft dezentral gespeichert und verwaltet werden. Die Komposition von Informationen aus verschiedenen Quellen stellt für das RDF kein Problem dar. Es entstehen lediglich größere Graphen. Würde man XML verwenden, hätte man Probleme bei der Zusammenführung der Informationen. Verschiedene Baumstrukturen zusammenzuführen, ist kompliziert und liefert oft nicht das gewünschte Ergebnis, da die sture Vereinigung von Bäumen keinen Baum darstellt und selbst verwandte Themen oft durch die verschiedenen Baumstrukturen getrennt werden.
Jede Aussage im RDF ist aus drei Einheiten aufgebaut: Dem Subjekt, dem Prädikat und dem Objekt. Der Zusammenschluss dieser drei Einheiten wird auch als *3-Tupel* (oder auch *Tripel*) bezeichnet.

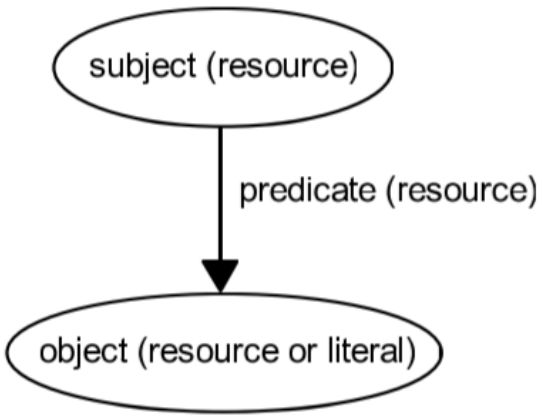

Abbildung 2.4.: RDF Statement (übernommen aus [Kar]).

Jede dieser Einheiten ist eine Ressource, wobei das Subjekt vom Prädikat näher beschrieben wird. Das Objekt kann wiederum eine Ressource sein, die in nächster Instanz auch als Subjekt verwendet werden kann.
Zwar ist im RDF die Komposition von Daten leicht, jedoch birgt der Bezeichner einer Ressource ohne weiteres einige Probleme. So könnte es z.B. sein, dass zwei verschiedene Ressourcen entweder

die gleichen sind mit verschiedenen Bezeichnern oder, dass sie unterschiedlich sind mit dem gleichen Bezeichner. Um dies vorzubeugen werden Ressourcen über *Uniform Resource Identifier (URI)* definiert. Dabei ist eine URI ein eindeutiger Bezeichner für jede Ressource. URIs sind dabei eine Verallgemeinerung von Uniform Resource Locatorss (URLs), die zum individuellen Zugriff auf Online-Dokumente verwendet werden. Werden solche Online-Dokumente in RDF beschrieben, wird oft die URL genommen. Ein typisches Beispiel ist die URI der RDFS-Ontologie. Diese Ontologie kann durch die URL `http://www.w3.org/2000/01/rdf-schema#` aufgerufen werden. Diese URL ist auch gleichzeitig die URI der Ontologie. Auch zu beobachten ist ein „“ am Ende der URL. An diesem Hashtag werden dann die Namen der einzelnen Ressourcen angehängt. So ist `http://w3.org/2000/01/rdf-schema#subClassOf` der individuelle Bezeichner der Ressource *subClassOf*.

2.3.2.1. Syntax

Graphen sind zwar verständlich für den Menschen (wenn sie nicht all zu groß sind), jedoch zur Analyse für den Computer eher ungeeignet. Dazu kommt, dass bei Datensätzen mit mehreren 1000 Einträgen eine Visualisierung des Graphen für den Menschen unübersichtlich ist.
Es gibt verschiedene Möglichkeiten zur Serialisierung von RDF-Dokumenten. Die gängigste und verbreitetste Variante ist die XML-Serialisierung. Diese Schreibweise kann nämlich mit den gängigen XML-Parsern leicht eingelesen und verarbeitet werden. Der vorher genannte Unterschied der Datenmodelle (Bäume vs. Graphen) ist hier kein Hindernis, da mit XML nur die Syntax vorgegeben wird, die den RDF-Graphen kodiert. Jedoch muss die Kodierung der RDF-Tripel hierarchisch erfolgen, da XML hierarchisch aufgebaut ist [al08].
Hierbei werden die Tripel nach dem Subjekt sortiert. Ein Beispiel von P.H. et al. verdeutlicht den Ansatz [al08]:

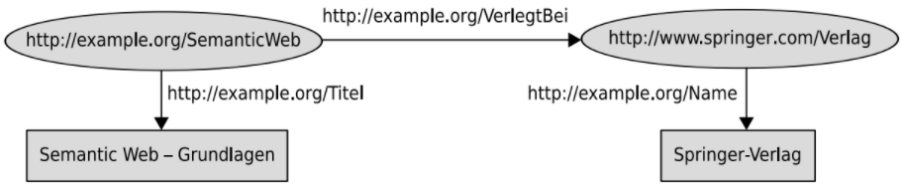

Abbildung 2.5.: RDF Beispiel Graph (übernommen aus [al08])

Abbildung 2.5 beschreibt ein Beispiel Tripel mit weiteren *Literalen*. Im Gegnsatz zu den URIs werden Literale als Rechtecke dargestellt. Bei Literalen handelt es sich um Datentypen (String, int, double, usw.). Die gängigste Variante von Literalen in Ontologien sind Strings [al08].

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3        xmlns:ex="http://example.org/">
4
5     <rdf:Description rdf:about="http://example.org/SemanticWeb">
6        <ex:Titel>Semantic Web - Grundlagen</ex:Titel>
7        <ex:VerlegtBei>
8           <rdf:Description rdf:about="http://springer.com/Verlag">
9              <ex:Name>Springer-Verlag</ex:Name>
10          </rdf:Description>
11       </ex:VerlegtBei>
12    </rdf:Description>
13 </rdf:RDF>
```

Listing 2.1: Kodierung zu Abbildung 2.5

Zeile 1 im Code 2.1 ist, wie im vorherigen Kapitel beschrieben, optional und nennt die XML-Version und deren Kodierung. Anschließend folgt der Anfang des Dokuments mit dem Wurzelknoten **rdf:RDF**. Dabei sind **ex:** und **rdf:** sog. Namensräume, die deklariert werden. Namensräume legen fest, welche Elemente in dem Dokument verwendet werden und werden als Werte des Attributes „*xmlns*“ angegeben. Innerhalb des **rdf:RDF** wird das Tripel beschrieben. Subjekt und Objekt werden jeweils mit dem **rdf:Description** beschrieben, während dessen Bezeichner durch das Attribut **rdf:about** angegeben werden. Die Literale der jeweiligen Ressourcen werden als Inhalt des Prädikats vorgestellt, das als Element **ex:VerlegtBei** dargestellt wird.

2.3.2.2. RDF Scheme

Ontologien beschreiben jedoch nicht nur die Beziehungen einzelner Individuen, sondern komplette Domains mit Typen und Klassen. Steht uns ein bestimmtes Vokabular zur Verfügung (ein Satz von Bezeichnern für die Entitäten), ist uns klar wie diese grob im Zusammenhang stehen. Während für Menschen einige Schlüsse gut nachvollziehbar sind, müssen die Zusammenhänge der einzelnen Bezeichner dem Computer in irgendeiner Form mitgeteilt werden. So versteht jeder Mensch, dass ein Professor ein Mitarbeiter der Uni Stuttgart ist und die Bahnlinie S2 nicht. Die S2 hält an der Universität und hat dort eine eigene Station. Für den Computer sind das ohne Weiteres nur Zeichenketten ohne Verknüpfungen.
Um dem Computer sog. *terminologisches Wissen* (Hintergrundinformationen) beizubringen, wurde das RDF-Schema (RDFS) entwickelt. RDFS ist dabei ein spezielles Vokabular dessen Namensraum ausgeschrieben *http://www.w3.org/2000/01/rdf-schema* und abgekürzt *rdfs* lautet. Der Sinn hinter dem Vokabular ist, universelle Ausdrucksmittel bereitzustellen, die innerhalb des Dokuments Aussagen über semantische Beziehungen ermöglichen. RDFS ermöglicht es Schemawissen zu spezifizieren und Abhängigkeiten zwischen Ressourcen zu beschreiben. Aus diesem Grund ist RDFS auch eine Wissenrepräsentationssprache. RDFS ist jedoch nicht wirklich ausdrucksstark und wird deshalb nur zur Beschreibung von lightweight Ontologien benutzt. [al08]
Um implizites Wissen darstellen zu können, benötigt man Repräsentationssprachen, die auf formaler Logik basieren (wie RDFS). Zwar lassen sich mit dem RDFS einfache Ontologien modellieren, jedoch reicht die Ausdruckskraft bei komplexeren Zusammenhängen nicht aus. Die ist nämlich beabsichtigt sehr limitiert gehalten. Beispele aus [Sic14] sind:

- **Lokaler Scope von Properties**: Man kann bei Properties keine Einschränkungen der Reichweite definieren, die nur für einige Klassen gilt. So kann man beim Beispiel des Properties *essen* nicht einschränken, dass Kühe nur Gras, während andere Tiere auch Fleisch essen.
- **Disjointness von Klassen**: In RDFS ist es nicht möglich zu sagen, dass zwei Klassen überschneidungsfrei sind. So sind die Klassen *ungerade Zahl* und *gerade Zahl* überschneidungsfrei, aber man kann sie beispielsweise nur als Subklasse von *Zahl* darstellen.

2.3.3. Web Ontology Language: OWL

Zwar ist die Ausdruckskraft in Ontologien ein wichtiges Mittel, um Zusammenhänge darzustellen, jedoch erhält man bei höherer Ausdruckskraft gleichzeitig eine ineffizientere Argumentationsunterstützung. So kann sie so ineffizient sein, dass es für die Praxis unberechenbar ist. Es gilt also, einen Kompromiss im Tradeoff zwischen den beiden Attributen zu finden.
Um dieses Problem zu lösen, wurde *OWL* entwickelt mit den drei verschiedenen Subsprachen *OWL-Full*, *OWL DL*, *OWL Lite*. Jede dieser Subsprachen hat Vor- und Nachteile, die in Abbildung **??** kurz aus [al08] zusammengefasst werden. Ontologienentwickler entscheiden sich dann für die Variante, die am besten zu ihren Bedürfnissen passt.

OWL FULL	**OWL DL**	**OWL Lite**
• Enthält OWL DL und OWL Lite • Enthält ganz RDFS • Sehr ausdrucksstark • Unentscheidbar • Wird durch aktuelle Softwarewerkzeuge nur bedingt unterstützt	• Enthält OWL Lite und ist teilsprache von OWL FULL • Entscheidbar • Wird von aktuellen Softwarewerkzeugen fast vollständig unterstützt • Komplexität NExpTime (worst case)	• Teilsprache von OWL DL und OWL FULL • Entscheidbar • Weniger Ausdrucksstark • Komplexität ExpTime (worst case)

Tabelle 2.1.: OWL Subsprachen im Überblick (Informationen aus [al08])

2.3.3.1. Syntax

- **Header:** jedes OWL-Dokument ist auch ein RDF Dokument und wird auch OWL-Ontologie genannt. Aufgrund dessen besteht ein OWL-Dokument am Anfang aus einem Header, mit dem *rdf:RDF* Element, um eine Anzahl an Namensräumen zu spezifizieren. Des Weiteren kann eine *Collection* folgen, die Kommentare, Version control und Inklusionen weiterer Ontologien beinhaltet. Diese sind eingebettet im Element *owl:Ontology*. So könnte, aus [Sic14] übernommen, ein Beispiel folgendermaßen aussehen:

```
<!--------- HEADER ---------->

<rdf:RDF
    xmlns:owl  =" http:  // www.w3.org /2002/07/ owl#"
    xmlns:rdf  =" http:  // www.w3.org /1999/02/22- rdf - syntax -ns#"
    xmlns:rdfs =" http:  // www.w3.org /2000/01/ rdf -schema#"
    xmlns:xsd  =" http:  // www.w3.org /2001/ XLMSchema#">

<!--------- Collection ---------->

<owl:Ontology  rdf:about =""">
    < rdfs:comment >An example  OWL ontology</ rdfs:comment >
    <  owl:priorVersion
        rdf:resource  =" http:  // www.mydomain.org/ uni -ns- old" />
    < owl:imports
        rdf:resource  =" http:  // www.mydomain.org/ persons " />
    <  rdfs:label  > University  Ontology </  rdfs:label  >
</ owl:Ontology >
```

Listing 2.2: Beispiel von einem Header und einer Collection in OWL

- **Klassen, Properties und Individuen**: Wie in RDFS bilden Klassen und Propertys die Grundbausteine des Dokuments. Klassen werden hierbei durch das *owl:Class* Element definiert. Dabei ist *owl:Class* eine Subklasse vom Element *rdfs:Class*. Beispiel 2.3 zeigt, wie ein einfaches RDF Tripel erzeugt werden kann:

```
<!--------- RDF Tripel ---------->

<  rdf:Description     rdf:about =" Fußballspieler  ">
    < rdf:type     rdf:resource  ="&owl; Class " />
</  rdf:Description  >


<!--------- Kurzform ---------->

< owl:Class    rdf:about =" Fußballspieler  " />
```

Listing 2.3: Beispiel für ein RDF Tripel mit Kurzform in OWL

Mit dem Identifier *rdf:about* wird der OWL-Klasse der Fußballspieler zugewiesen. Statt *rdf:about* kann auch *rdf:ID* verwendet werden. So können Beziehungen zwischen Klassen in der Definition miteingefügt werden, wie z.B. *owl:disjointWith* oder *owl:equivalentClass*.
In OWL gibt es zwei verschiedene Arten von Propertys. Zum einen die *Object properties* und zum anderen die *Datatype properties*. Beide sind Subklassen von *rdf:Property*. Object properties beschreiben die Beziehungen zwischen den Objekten (z.B. *arbeitetIn* ...), während Datatype properties Objekte an sich genauer beschreiben bzw. Objekte mit Werten verknüpfen. Des Weiteren kann man bei Properties gewisse Einschränkungen definieren. Der Beispielcode 2.4 sagt aus, dass Fußballspiele nur von Fußballspielern gespielt werden können.

```
1 < owl:Class    rdf:about =" Fußballspiele ">
2     < rdfs:subClassOf >
3         < owl:Restriction >
4             < owl:onProperty    rdf:resource =" werdenGespieltVon "/>
5             < owl:allValuesFrom    rdf:resource =" Fußballspieler "/>
6         </ owl:Restriction >
7     </ rdfs:subClassOf >
8 </ owl:Class >
```

Listing 2.4: Beispiel für Einschränkungen von Properties in OWL

owl:subClassOf spezifiziert, dass Fußballspiele unter dem Property *werdenGespieltVon* eine Subklasse von Fußballspielern ist und somit jede Instanz davon unter dem Property auch eine Instanz von Fußballspielern ist. *owl:allValuesFrom* und *owl:onProperty* definieren, dass jede Instanz von dieser Property aus der Klasse Fußballspiele kommen muss. Dementsprechend dürfen Fußballspiele nur von Fußballspielern gespielt werden.
Im Syntax von OWL gibt es neben dem hier erklärten Vokabular noch viel mehr, das jedoch aus Gründen des Umfangs nicht weiter vorgestellt werden kann.

3. Computer-Aided Engineering (CAE)

Seit jeher gibt es in der Industrie den Kampf um das beste Produkt. Dafür muss das Produkt kosteneffizient und qualitativ hochwertig sein. Mit dem Informationszeitalter und den damit kommenden Computern, wurde eine neue Facette der Qualitätssicherung und des kostengünstigen Produktentwurfs erschaffen. *Computer-Aided Engineering (CAE)* beschreibt dabei die Technologie, Simulationen aus *Computer-Aided Design (CAD)*-Modellen durchzuführen, sie zu analysieren, um so Erkenntnisse über das Produkt unter bestimmten Bedingungen zu erfahren.
Während beispielsweise früher in der Autoindustrie mit realen Prototypen Crash-Tests simuliert wurden, übernehmen heutzutage Computer diese Simulation. So lässt sich mithilfe von Crashsimulationen neben Beträgen in Millionenhöhe für die Prototypen-Erstellung auch eine Verkürzung der Entwicklungszeit um mehrere Monate einsparen [13]. Auch ist man bei den Crashsimulationen variabler, da die Umgebungsparameter leicht verändert werden können (zum Beispiel die Dicke der Mauer, die Materialeigenschaften, Geschwindigkeit, usw.).
Computerbasierte Simulation lässt sich grobgranular in die Bereiche CAD und CAE unterteilen. CAD stellt dabei 3D-Modelle von Produkten zur Verfügung, während im CAE diese Produkte analysiert werden, um so ihre Eigenschaften zu untersuchen. Die CAD Modelle können dabei jedoch nicht ohne Veränderungen in den Simulationen genutzt werden, da CAE Daten noch umfangreicher und meistens aus mehreren CAD Modellen gleichzeitig aufgebaut sind. So sind auch Randbedingungen der Geometrien, Materialdaten, Parameter und Lösungsalgorithmen enthalten [al20c].
Um eine Ontologie einer Domäne zu konstruieren, müssen detaillierte Kenntnisse der Domäne vorhanden sein. Ein weiterer wichtiger Teil der Ausarbeitung galt der Auseinandersetzung der CAE-Domäne und dessen Beschreibung. Dabei fokussiert sich dieses Kapitel auf die Erläuterung der CAE-Domäne am Beispiel einer standardisierte und typischen Integration von CAE-Simulationen in der Automobilindustrie [13] und deren genauen Teilprozesse.

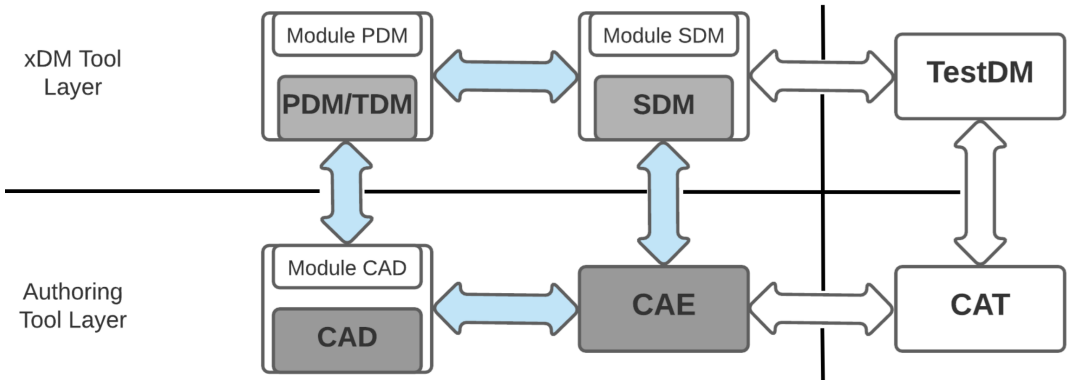

Abbildung 3.1.: Computer Simulation im Überblick (übernommen aus [Ste09])

Wie bereits erwähnt, ist die Computersimulation in der Autoindustrie nicht mehr wegzudenken. Abbildung 3.1 zeigt einen Überblick über den Ablauf eines standartisierten Simulationsprozesses in der deutschen Autoindustrie und wie CAE in einem solchen Prozess integriert wird. Dabei wird generell zwischen zwei Layern unterschieden. Zum einen das *xDM Tool Layer*, das für das Datenmanagement und zum anderen das *Authoring Tool Layer*, das für die Manipulation und Simulation der Daten zuständig ist. Aus einem Layer korrespondiert ein Modul mit einem aus dem anderen Layer, die zusammen entweder für Design (CAD, PDM/TDM), Simulation (CAE, SDM) oder physikalisches Testen (CAT, TestDM) zuständig sind.
Die Module im xDM Tool Layer stellen strukturierten Zugriff auf Informationen bezüglich Entwicklungsprozessen, Kontrollprozessen, Koordination sowie Datenspeicher und Administration zur Verfügung. Im Kontext der CAD/CAE Integration sind es die Module *Product Data Manager (PDM), Team Data Manager (TDM) und Simulation Data Manager (SDM)*. Den Erklärungen von Stefan Bauer et al. übernommen, haben die Module folgende Funktionen [Ste09]:

3.1. Product Data Manager / Team Data Manager (PDM/TDM)

Das PDM-Modul stellt sämtliche Funktionen zur Verfügung, die zur Administration der Daten im Entwicklungsprozess benötigt werden. Darunter gehören technische Beschreibungen der Produkte, die Fähigkeit, die Fortschritte des Projekts zu illustrieren, diese auch über verschiedene Entwicklungsprozesse hinweg. Das PDM ünterstützt auch Collaboration Engineering, also die Kooperation verschiedener Partner (-Firmen) in der Entwicklung. Dies reicht vom Datentransfer der Partner bis hin zur Entwicklung von Prototypen.
Das TDM hingegen ist primär für die toolspezifischen Kontrolle des Informationsmanagements und der Datenformate zurständig, u.a. für die die CAD-Geometrie und der technischen Dokumentation. Je nach Größe der Firma kann das PDM funktionelle Teile der TDM übernehmen.

3.2. Simulation Data manager (SDM)

Wie in Abbildung 3.1 zu sehen ist, steht das SDM in enger Verbindung mit CAE. Der Prozess hier ist durch die Verschiebung aus dem physikalischem Prototyping ins virtuelle Prototyping gekennzeichnet und benötigt dementsprechend ein strukturiertes Datenbankmanagementsystem für die hier anfallenden Simulationsdaten. Das SDM hat neben der Verwaltung der Simulationsdaten auch die Aufgabe, diese effektiv zu suchen und zu finden. Aufgrund der in der Simulation rapide steigenden Anzahl an heterogenen Simulationsdaten stellt dies eine Herausforderung dar. Meistens übernimmt das SDM die Rolle des TDM für die CAE-Abteilung und ist einem größeren PDM System unterstellt Es kombiniert die grundlegenden Funktionalitäten des Datenmanagements und des Apllication Process Controls. Die Tabelle zeigt u.a. untergeordnete Aufgaben der grundlegenden Funktionalitäten:

Datenmanagement	Application Process Control
- Entwicklung und Instandhaltung einer Produktstruktur	- CAE Workflow Management (Nachvollziehbare Ausführung und Dokumentation der Prozesse)
- Verständliches und Nachvollziehbares Management von Produktdaten (auch für Inputdaten aus anderen Domains (z.B. CAD, testing,...))	- Process Control & Integration (Miteinbeziehung von Simulationsinformationen in den Entwicklungsprozess des Produkts)
- Organisation und Speicherung der Simulationsdaten inklusive ihrer Beschreibungen (Metadaten)	
- Ständige Erweiterung und Versionierung des Entwicklungsverlaufs	

Tabelle 3.1.: Detailliertere Aufgaben des SDM in den Bereichen des Datenmanagements und Application Process Conrtol

3.3. Computer-aided Design (CAD)

CAD beschreibt die Technologie, Computersysteme in der Herstellung von Produkten, deren Manipulation, Analysis und Optimierung zu benutzen. Dabei wird jedes Computerprogramm, das Computergrafiken und ein Anwendungsprogramm enthält, das technische Funktionen im Entwurfsprozess erleichtert, als CAD-Software klassifiziert [Lee99]. Dementsprechend variiert die Form der Design Produkte von 2D Zeichnungen und 3D Modellierungen über topologische Verbindungen bis hin zu Parameterdaten (Material, Dicke,...) und Konzeptdaten aus der Entwicklungsphase.
So beschreibt Abbildung 3.2 den groben Prozess einer üblichen CAD-Entwicklung. In erster Instanz gibt es das Bedürfnis nach einem Produkt oder nach einer Lösung für ein Produkt. Diese greift in die Spezifikation über, die dafür zuständig ist, das Problem technisch gesehen zu definieren und dessen Bedürfnisse und Bedingungen zu veranschaulichen. Aus dem Schaubild lässt sich auch konstatieren, dass es sich bei diesem Prozess um einen Kreislauf handeln kann. Dies hat den Hintergrund, dass während des Designs oft erkannt wird, dass Probleme und die Bedürfnisse an das Produkt neu definiert werden müssen. Grobe Sketches helfen dabei, die Kreativität zu steigern und neue Ideen mit einzubringen.
In der Synthese werden bestimmte Charakteristiken bezüglich des Produkts gewählt. Auch hier handelt es sich um einen Kreislauf. Befindet sich der Prozess am Anfang, so dient dieser Schritt auch als Ideenfindung. Besteht jedoch schon eine gewisse Wissensbasis über das Produkt, so dient die Synthese als Optimierung oder falls das Produkt schon optimal mathematisch definiert ist, auch als Suche für eine bessere Alternative. Hier werden auch sog. CAD-Preprocessors integriert, um dreidemensionale Modelle zu entwickeln und den Schritt der Analyse vorzubereiten. Damit können bestimmte Tools unterstützt werden.

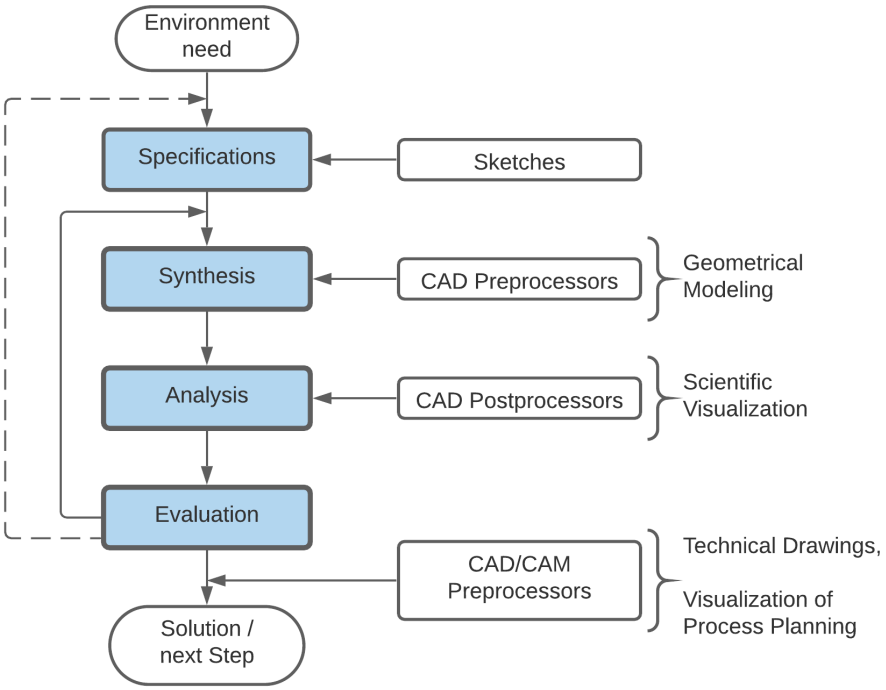

Abbildung 3.2.: Üblicher CAD Prozess (basiert auf [COm20])

Anschließend wird das Design bezüglich der Umwelt und der Bedingungen analysiert, um die Qualität und die Befriedigung der Bedürfnisse zu bestimmen. Im CAD-Postprocessor werden dementsprechend wissenschaftliche Visualisierungen (Graphen, Statistiken, Hervorheben von Fehlern, ...) entwickelt, die dem Entwickler bei der Interpretierung der Ergebnisse und der Evaluation der Lösung am Ende behilflich sein sollen.
Mithilfe dieser Daten wird das Produkt evaluiert, ggf. erneut im Spezifizierungs- oder Synthese Teil aufgegriffen und korrigiert. Erfüllt das Produkt die vorgegebenen Bedingungen, so wird es mithilfe von CAD/CAM-Preprozessoren detailliert in eine standartisierte Sprache transformiert und für den nächsten Schritt freigegeben. Computer Aided Manifacturing (CAM) beschreibt dabei die Benutzung von Computer Technologie (z.B. CNC-Maschinen) zur Planung und Kontrollierung der Produktherstellung mithilfe der Ergebnisse des CAD Prozesses. Die Analyse und Evaluation sind Aufgabengebiet des CAE.

3.4. Computer-Aided Engineering (CAE)

Damit CAD-Modelle zu simuliert werden können, müssen die Geometrien ggf. transformiert werden, um für die CAE Tools überhaupt simulierbar zu sein. Dieser Schritt wird im Prozess als Preprocessing bezeichnet und ist ein großer Bottleneck in der Automatisierung vom Simulation Workflow. Neben der eigentlichen Rechenzeit der Simulation, müssen oft CAD-Modelle mit mehreren Tausenden Komponenten in CAE-Modelle umgewandelt werden [al20a]. Hinzu kommt, dass ein Modell oft in

mehrere CAE-Modelle für verschiedene Arten von Simulationen umgewandelt wird. So beschreibt Abbildung 3.3 den üblichen CAE-Prozess mit seinen einzelnen Schritten. Die einzelnen Schritte werden aus [Ste09] erklärt.

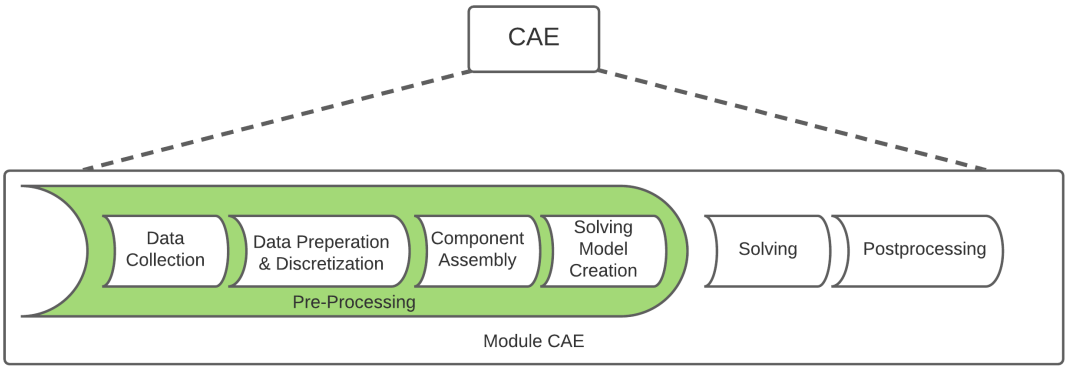

Abbildung 3.3.: Gängiger CAE-Prozess (übernommen aus [Ste09])

Grob aufgeteilt besteht der CAE-Prozess aus dem Preprocessing, der eigentlichen Simulation und dem Postprocessing. Wie bereits erwähnt ist das Preprocessing aufwändig und hier auch in den einzelnen Teilschritten *Data Collection, Data Preperation & Discretization, Component Assembly* und *Solving Model Creation* aufgeteilt.
Data Collection kümmert sich um den Import und der Qualitätsprüfung der Daten. So kann es sich hierbei um beispielsweise CAD Modelle oder andere Ergebnisse von schon durchgeführten Simulationen handeln. Es werden u.a. Daten von verschiedenen Abteilungen und Datenbanken gesammelt und anschließend für den weiter laufenden Prozess im SDM gespeichert. Hierbei werden keinerlei Daten manipuliert, noch in irgendeiner Form organisiert.
Die Data Preparation & Discretization umfasst jegliche Aufgaben zur Manipulation der Daten, damit sie in der Simulation verwendet werden können. Inwieweit die Daten manipuliert werden müssen, hängt dabei von den Daten selbst ab. So müssen z.B. für CAD-Modelle die Geometrien vereinfacht, Randbedingungen abgeleitet, Attribute & Parameter ergänzt und Verbindungsinformationen der CAD Modelle bestimmt werden.
Sind die Daten diskretisiert, so kann es sein, dass mehrere CAD-Modelle ein ganzes CAE-Modell bilden sollen. Im Component Assembly werden diese Modelle in einem Ganzen zusammengebaut. Dieser Schritt ist optional und findet nur im genannten Fall der mehrteiligen Komponenten statt.
Im letzten Schritt (Solving Model Creation) des Preprocessings wird ein Lösungsmodell erstellt, in dem alle für die Simulation benötigten Daten zusammengestellt werden. Das Ergebnis ist ein Input-Deck mit spezifisch angepassten Parametern, das vom Solver benutzt werden kann, um Simulation durchzuführen. Auch hier werden alle manipulierten Daten im SDM gespeichert.
Ist das Preprocessing abgeschlossen und sind die Daten für die Simulation vorbereitet, so folgt die eigentliche Simulation (Solving). Hierbei werden das vorher vorbereitete Input-Deck geladen, Parameter angepasst, Vorberechnungen durchgeführt, Berechnungsoptimierungen verrichtet und letztendlich die eigentliche Simulation eingeleitet.
Das anschließend folgende Postprocessing entspricht der Evaluation im vorherigen CAD-Modell. Dies beinhaltet die Überprüfung der Gültigeit der Simulation, den Vergleich der Ergebnisse mit den gemessenen Daten, die Visualisierung anhand von Tabellen oder Grafiken, die Erstellung von Be-

richten, die Umwandlung der Ergebnisse in andere Formate (z.B. PDF, ...) und die Schlussfolgerung der Simulation. Wie in jedem vorherigen Schritt werden diese Informationen im dafür zuständigen SDM gespeichert.

3.5. CAD-CAE Workflows

Wie bereits erwähnt, werden die CAD-Modelle für die CAE-Simulation ggf. verändert. Dies ist auch davon abhängig, um welche Art von Simulation oder Berechnung handelt. Möchte man Simulationen mit Flüssigkeiten durchführen, so wird eine andere CAE-Applikation benötigt, als wenn Deformationen von Materialien simuliert werden wollen. Das folgende Kapitel aus [al13] beschreibt gängige CAD-CAE Workflows in der Automobilindustrie und zeigt u.a. den direkten Datenaustausch mit dem CAD-System.

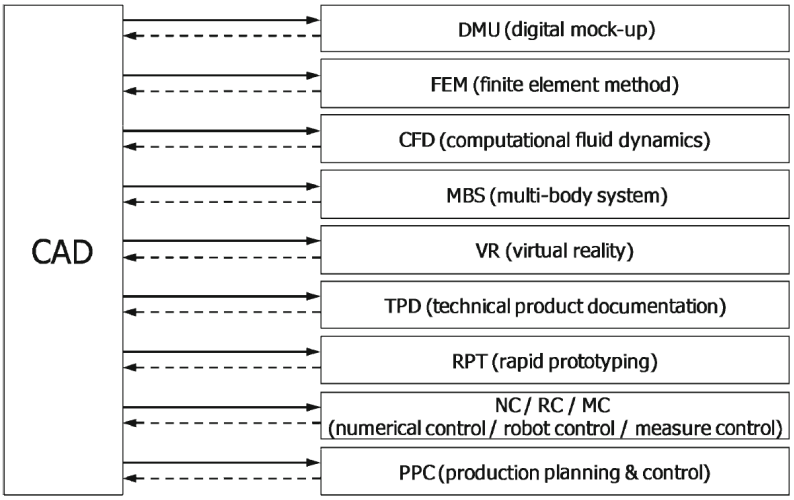

Abbildung 3.4.: CAD/CAE Workflow in der Automobilindustire (übernommen aus [al13])

Abbildung 3.4 zeigt dabei den Überblick der verschiedenen Arten von CAD/CAE-Workflows in der Automobilindustrie. Die durchgezogenen Pfeile verdeutlichen, dass jedes dieser CAE Applikationen mit dem CAD-System direkt verbunden ist. Je nachdem, um welche Applikation es sich handelt, wird das 3D CAD-Master Modell konvertiert und in die zuständige CAE-Umwelt transferiert. Nach der Evaluation werden die Ergebnisse rückwirkend auf das CAD Modell benutzt, um es nachhaltig zu verbessern und zu optimieren (gestrichelter Pfeil).

3.5.1. DMU

In der Regel werden die CAD-Modelle erweitert für die anstehende Simulationen. DMUs sind jedoch vereinfachte geometrische Repräsentationen eines Produkts . Sie werden auch *digitag dummies* gennant. DMU-Prozesse werden für Kollisionserkennung, Packaging-Studien, Mounting und Assembling benutzt. Der Datentransfer erfolgt unkompliziert über z.B. neutrale Datenformate wie

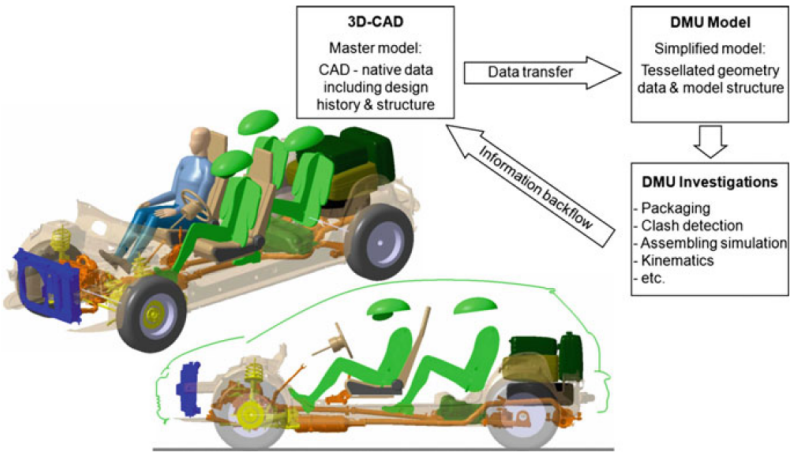

Abbildung 3.5.: DMU Beispiel in der Automobilindustire (übernommen aus [al13])

Standard for the exchange of product model data (STEP). Die Genauigkeit der Modelle ist durch die Reduzierung des CAD-Modells auch geringer als im CAD-Modell an sich und eine Modifizierung des DMU ist in der Regel nicht möglich.

Abbildung 3.5 zeigt ein Beispiel eines DMU-Workflows anhand eines Autos. Es lässt sich beobachten, dass das DMU-Modell deutlich informationsarmer ist und nur grobe Informationen und Platzhalter für z.B. Reifen beinhaltet. Hierbei diente das DMU-Modell als Grundlage für verschiedene Styling-Vorschläge für das Auto mit Berücksichtigung der geometrischen Anforderungen von Komponenten- und Ergonomiekonfigurationen.

3.5.2. FEM

Die FEM teilt CAD-Modelle in endlich viele Bereiche auf, die jeweils mit festgelegten Nodes verbunden sind. Jedem dieser Bereiche kann man anschließend bestimmte Materialeigenschaften zuordnen. Ziel dieser Methode ist das Modell unter Stress, Deformation, thermischer Belastung oder NVH (noise, vibration und harshness) zu testen. Je nach Geometrie und nach Art des Tests werden verschiedene Herangehensweisen benutzt.

Abbildung 3.6 beschreibt dabei den groben Prozess, welcher jedoch immer stattfindet. Am Anfang werden die Daten des CAD-Modells durch einen diskretisierten Prozess zu der FEM-Applikation transferiert. Diese verändert anschließend das Modell und teilt es in kleinere Geometrien auf (*meshing*). Die Geometrien können ein-, zwei-, oder dreidimensional sein. Anhand des gemeshten Modells finden die FEM-Berechnungen statt, um Randbedingungen, Materialeigenschaften und bestimmte Vorbedingungen oder Regeln miteinzubeziehen. Erst danach folgen die Analyse und die anschließende Evaluation rückführend, um die Ergebnisse der Simulationen in ein optimierteres CAD-Modell miteinfließen zu lassen.

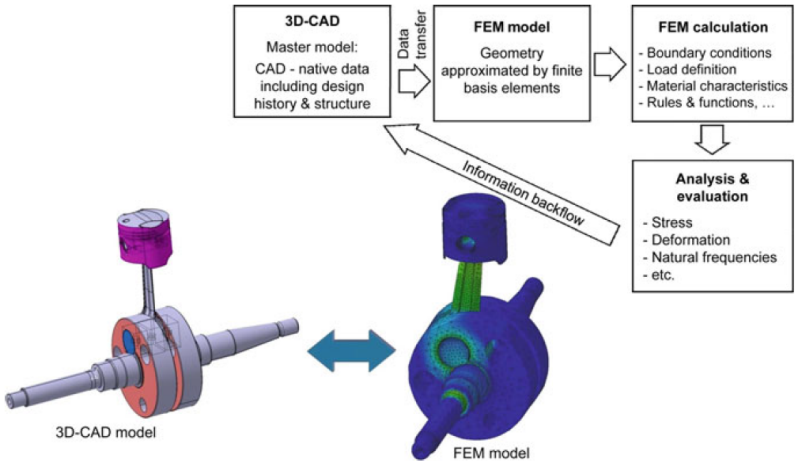

Abbildung 3.6.: FEM Beispiel einer 1-Zylinder Kurbelwelle in der Automobilindustire (übernommen aus [al13])

3.5.3. CFD

CFD ermöglicht qualitative und quantitative Simulationen und Vorhersagungen von Flüssigkeitsverhalten. Ähnlich wie bei FEM werden die CAD-Modelle in approximierte Geometrien gemesht. Diese Diskretisierung kann auf verschiedenen Leveln der Komplexität erfolgen. So sind eindimensionale Simulationen in einer Röhre nicht so kompliziert wie Dreidimensionale Simulationen z.B. im Motorraumfluss. Die CAD-Daten werden wie beim DMU mit STEP transferiert und nach Einbettung der Rahmenbedinugnen und Eigenschaften können die Simulation und die anschließend anstehende Analysis und Evaluation angegangen werden. Abbildung 3.7 zeigt dies anhand eines Motors. Für eine genauere Erklärung und einer Erklärung der anderen CAD/CAE Workflows ist auf die Originalquelle zu verweisen.

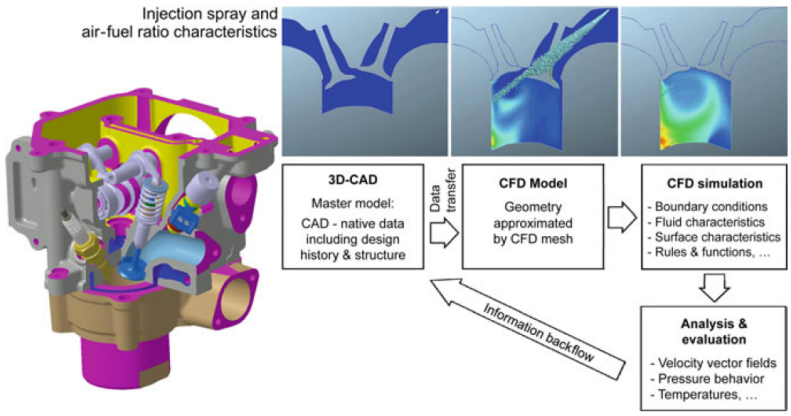

Abbildung 3.7.: CFD Beispiel anhand eines Motors (übernommen aus [al13])

3.6. Computer Aided Testing (CAT)

sind meistens Anwendungen, die die Testtechniken steuern, um die Qualität von den zu testenden Produkten zu bewerten. Hierbei wird bewertet, ob das Produkt unter bestimmten Bedingungen unter den gegeben Toleranzen liegt oder nicht. Abbildung 3.8 zeigt hierbei anhand von TESSY [al20b], einem CAT Applikations Tool, welche für Aktivitäten zu einer solchen Applikation gehörten.

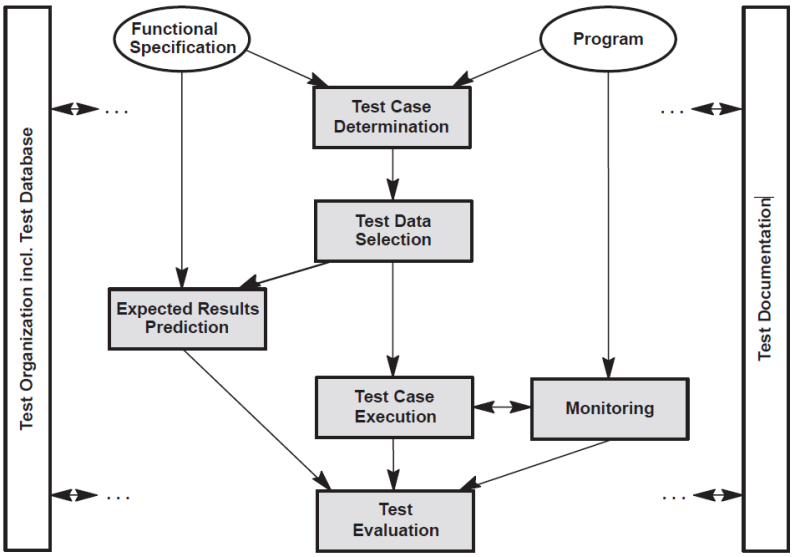

Abbildung 3.8.: CAT Aktivitäten (übernommen aus [al20b])

Wie jedes vorherige Modul benötigt auch dieses eine Datenbank. In Abbildung 3.1 gekennzeichnet durch TestDM und hier als *Test Organization*. Hier enthalten ist das ganze Management von den zu testenden Objekten inklusive der dafür anfallenden Daten. Dazu gehören z.B. die Testfälle, die Testdaten, die zu erwartenden Ergebnisse und die technischen Einstellungen.
In der *Test Case Determination* werden die Tests für die jeweiligen Objekte genau definiert. Dabei ist ein Test Case eine bestimmte Simulation von einem Objekt mit bestimmten Eingabedaten. Dadurch, dass durch die Test Case Determination die Qualität und der Umfang der Tests bestimmt wird, ist dieser Schritt auch der wichtigste.
Anschließend werden in der *Test Data Selection* die bestimmten Daten für die einzelnen Test Cases ausgewählt. Dabei muss der Tester für jeden Test einen Satz von Input-Variablen aussuchen, mit denen das Objekt getestet werden soll. Zu jedem Test Case werden die erwarteten Ergebnisse abgeleitet (*expected results prediction*). Sind keine genaueren Ergebnisse vorherzusagen, so wird durch das Verhalten, der Akzeptanzkriterien oder der Referenzdaten versucht, ein Ergebnis herzuleiten.
Sind die Daten ausgewählt, folgt mit ihnen die eigentliche Ausführung des Tests (*Test Case Execution*). Während der Ausführung wird das Verhalten untersucht und im *Monitoring* aufgenommen. Mithilfe des Monitorings, des laufenden Tests, deren und der vorhergesagten Ergebnisse, wird der Test Case evaluiert und analysiert, wie er im Vergleich zu den vorhergesagten Ergebnissen abgelaufen ist.
Zum Schluss werden diese Ergebnisse und deren Vergleich in der *Test Documentation*, auch für nicht im Testverlauf involvierte Personen, verständlich dokumentiert und sämtliche Vergleiche oder Fehlermeldungen zusammengefasst.

3.7. Anforderungen

Nach Betrachtung des Aufbaus einer Ontologie und der Charakterisierung der CAE-Landschaft, werden aus diesen gewonnenen Erkenntnissen die Anforderungen an die Ontologie abgeleitet und hier aufgelistet. Am Ende der Arbeit folgt eine Verifikation, um zu zeigen, ob und in wie fern diese Anforderungen erfüllt worden sind.

1) **Kompakte Abdeckung der CAE-Domäne:**
   So wurde bei der Betrachtung der CAE-Domäne klar, dass sie sehr breit gefächert ist. Kompakt bedeutet im Sinne der Ontologie, so viele Informationen in so wenig Klassen/Beziehungen/-Attributen wie möglich umzusetzen. So ist eine weitere Herausforderung diese weitgreifende Domäne kompakt, jedoch trotzdem ganzheitlich zu entwickeln. Gewisse Bereiche müssen ggf. abstrahiert werden.

2) **Konsistenz der Ontologie:**
   Damit die Ontologie korrekt ist, muss sie konsistent sein. Die Ontologie soll die CAE-Domäne beschreiben, wie sie auch in der Gegenwart repräsentiert wird. So sollten beispielsweise keine Instanzen einer Klasse auch möglicherweise Instanzen anderer Klassen sein, obwohl sie keinerlei Beziehung mit der Klasse beinhaltet.

3) **Generisches und erweiterbares Konzept einer Ontologie:**
   Wie in Abbildung 2.1 erwähnt, gibt es Ontologien, die eine spezifische Domäne beschreiben und Ontologien, die Domänen generisch beschreiben. Je nach Anwendungsgebiet, sind die jeweiligen Ontologiearten korrekt. Da es sich hier aber um eine breit gefächerte Domäne handelt, bedarf es hierbei eine generische Beschreibung. Viele Simulationstools, Firmen oder weitere Einheiten der Domäne, arbeiten sehr individuell und spezifisch. Jeden spezifischen Bereich genau mit einzubringen und eine ontologische Beschreibung dessen zu entwickeln, wäre utopisch und somit nicht machbar. Infolgedessen stellt die generische Beschreibung möglichst aller Bereiche der Domäne eine weitere Herausforderung, jedoch eine wichtige Anforderung, dar. Sind jedoch gewisse Bereiche aufgrund ihrer spezifischen Art nicht generisch darzustellen, jedoch für die Ontologie trotzdem wichtig, müssen diese für differenziertere Umgebungen generisch erweiterbar/anpassbar entwickelt werden.

4) **Richtiges Maß zwischen Gegenstands- und Aufgabenbezogenheit:**
   Eine weitere Anforderung ist das Gleichgewicht zwischen der Gegenstands- und Aufgabenbezogenheit. Diese sind in Abbildung 2.1 ebenfalls genannte Kriterien zur Beschreibung einer Ontologie. Je nach Anwendungsgebiet, beschreibt eine Ontologie entweder eher sachlich oder aufgabenspezifisch. Das richtige Maß in Bezug dieser Domäne zu finden, ist eine weitere Herausforderung.

4. Verwandte Arbeiten

Ontologien sind weit verbreitet und finden in fast jedem Bereich Verwendung. Ein Beitrag von Flavien Boussuge et al. [**CAP**] soll das in Kapitel 3.4. erwähnte Pre-Processing mithilfe einer Ontologie beschleunigen. Viele Schritte finden im Pre-Processing manuell statt und und die Ontologie soll die Verbindung zwischen Design und Simulation vereinfachen, indem sie bestimmte Bereiche automatisiert (z.B. Modellierungsentscheidungen), Analysedaten effizient verwaltet und die Zusammensetzung des CAE-Modells beschleunigt. So wird auch oft eine gewisse Expertise benötigt, um ein CAE-Modell aus einem CAD-Modell zu entwickeln. Die Ontologie soll dieses Verständnis auch für nicht Spezialisten erleichtern, indem sie beispielsweise wichtige Beziehungen zwischen Instanzen hervorhebt oder Ähnliches.

Bei der Abbildung 4.1 handelt es sich um die besagte Ontologie. Jede Analyse hat ein Ziel *Simulation Intent*) oder ein Verhalten das es zu beobachten gilt. Im *Simulation Objective* werden diese Intentionen abstrakt beschrieben. Aus den gewünschten Zielen wird ein erstes theoretisches Modell erarbeitet (*Simulation Model*). Dieses Modell wird hinsichtlich der Constraints kosntruiert (z.B. welche Art von Analyse, aus welchen Materialen, usw.). Aus diesem Simulationsmodell wird ein sog. Zellmodell entwickelt. Dies teilt das Modell in verschiedenen Zellen ein, um den Zellen unterschiedliche Attribute zuzuweisen. Beispielsweise sind einige Bereiche des Modells nicht wichtig und werden deshalb mit einer geringeren *Meshdichte* versehen, um effizienter zu sein. Oder ihnen werden *Morphologien* zugewiesen. Morphologien sind dabei Konzepte, um die Form der Zelle zu beschreiben. Mithilfe von Beziehungen und Inferenzen kann das System anschließend wissen, wie sie diese Form verarbeiten soll.

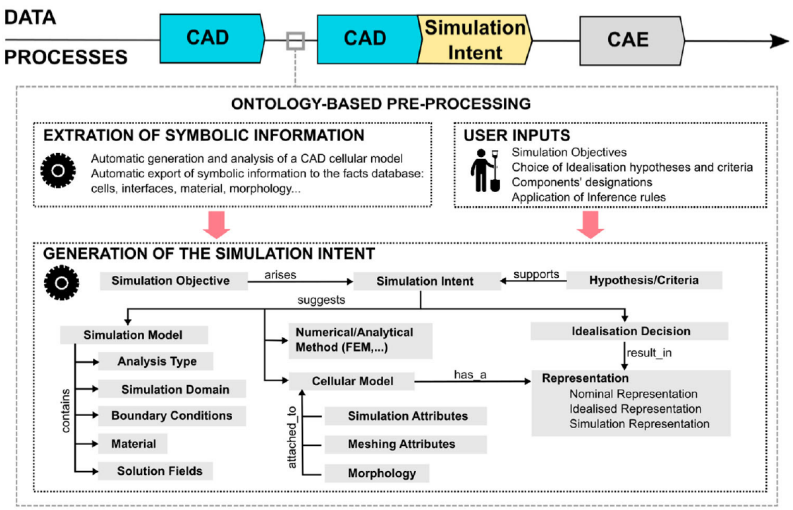

Abbildung 4.1.: Die Simulation Intent Ontology mit ihren Elementen und Beziehungen (übernommen aus [al20a])

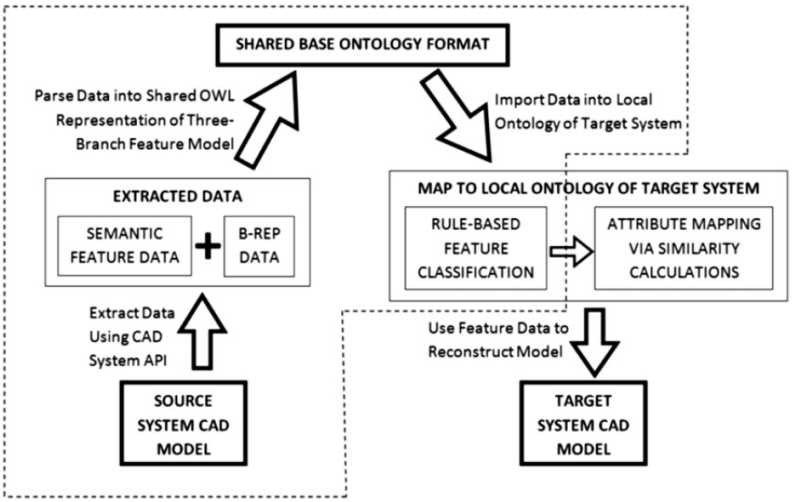

Abbildung 4.2.: Mapping zwischen Heterogenen Entwicklungssystemen (übernommen aus [TW13])

Diese Ontologie beschränkt sich jedoch auf thermo-mechanische Analysen und nur auf die Generierung von CAE-Modellen. Die hier entwickelte Ontologie erstreckt sich über einen weit aus größeren Bereich. Nützlich sind hier aber die Beschreibungen und mögliche Aufteilungen einzelner CAE-Modelle. Mit der Zeit werden CAD-Modelle komplexer und werden ggf. von mehreren Abteilungen gleichzeitig entwickelt. Eine weitere Ontologie von L. Sun und B. Ding [L S20] beschreibt eine Datenaustausch-Methode von heterogenen CAD-Daten. Die Ontologie stützt sich wie bei [**CAP**] auf ein Zellmodell des CAD-Modells. Die Geometrien werden als Zellen repräsentiert, die dabei Attribute (owner list) der jeweiligen Geometrie beinhalten.
Ähnlich beschreiben Sean Tessier und Yan Wang [TW13] eine Ontologie um CAD-Daten auszutauschen, jedoch zwischen heterogenen CAD-Entwicklungstools. Dieser Austausch zwischen den Systemen ist sehr kostspielig und zeitaufwendig und wird mithilfe von Ontologie basiertem mapping effizienter gestaltet. Wie in Abbildung 4.2 zu sehen, werden die Daten im Source-Modell erst extrahiert und in eine zwischen den Systemen gemeinsam genutzte Ontologie etabliert. Die extrahierten Daten beinhalten dabei jegliche relevanten semantische Informationen des Modells inklusiver *Boundary-Represantation Data (B-Rep Data)* für die Geometrie. Diese nun im Ontologie Format etablierten Daten werden vom Zielsystem importiert. Der Reasoner des Zielsystems untersucht die Eigenschaften der Daten, testet auf Fehler und versucht die Geometrie nachzubauen. Zum Schluss werden die erwähnten B-Rep Daten genutzt, um die Geometrie zu verifizieren und falls diese gleich sind, kann das Mapping gespeichert und die nächste Übersetzung ohne Testphase ablaufen. Auch hier ist es nützlich, Informationen über den Aufbau von CAD-Modellen abzuleiten und sie in die hier entwickelte Ontologie mit einfließen zu lassen. Jedoch beschreibt sie, wie bei Flavien Boussuge et al. [**CAP**], auch nur einen speziellen Bereich der CAE-Domäne. Dabei fehlt es an einer ganzheitliche Betrachtung des Bereiches und der darin enthaltenen Daten.

5. CAE-Ontologie

CAE-Daten bestehen jedoch nicht nur aus einem CAE-Modell und dessen Metadaten, sondern aus weiteren Datenpaketen, die u.a. Rohdaten, Spezifikation der Auftraggeber, detailliertere Testergebnisse usw. beinhalten. Dies wird zusammengefasst als CAE-Projekt beschrieben.
Um diese Metadaten zu explorieren, werden CAE-Daten benötigt. Vorliegender Arbeit wurden verschiedene Daten zur Verfügung gestellt. Diese beinhalten neben Testanalysen von verschiedenen Firmen auch deren Rohdaten und Beispiele für Auftragsanforderungen. Darüber hinaus stellt Siemens ein kostenloses Jupiter Tesselation (JT)-Tool zur Verfügung: *JT2Go*. JT2Go ist dabei ein 3D viewing tool für JT-Daten. JT2Go wird bei vielen weltweit führenden Fertigungsunternehmen für die Kommunikation, Visualisierung digitaler Modelle und eine Vielzahl anderer Zwecke verwendet [Off]. CAD-Modelle können im jt. Format gespeichert und mithilfe des Tools betrachtet werden. Dabei lassen sich die Eigenschaften des CAD-Modells ablesen und somit Metadaten ableiten.
Dadurch, dass CAE-Modelle komplexer sind als CAD-Modelle, wird auch dafür ein Tool benötigt. Die meisten Simulationstools sind jedoch kostenpflichtig. Nach Anfragen wurde von SolidWorks (zu ihrem Simulationstool) ein Modulhandbuch zur Verfügung gestellt. Solidworks ist ein vom französischen Unternehmen *Dassault Systèmes* erstelltes Programm, welches CAD- und CAE-Modelle zu simuliert. In diesem Handbuch wird erklärt, wie mithilfe von CAE-Modellen bestimmte Simulationen durchgeführt werden. Dementsprechend werden viele Beispiele illustriert, welche die Ableitung von Metadaten ermöglichen.
Die Ontologie wird im Rahmen dieser Arbeit in Form eines Modells und der dazugehörigen OWL-Datei präsentiert. In erster Linie wird das Modell erklärt, dessen Aufbau und aus welchen Daten welche Klassen in der Ontologie entstanden sind. Als nächstes wird die Implementation der Ontologie in der dazugehörigen Ontologie-Sprache OWL schematisch erläutert. Diese befindet sich in der Arbeit aufgrund der Länge als externe OWL-Datei. Hierbei wird an beispielhaften Snippets die Syntax und die Semantik der Datei erklärt. Eine genaue Erklärung wäre zu umfangreich v.a., da sich viele Klassen, Attribute und Beziehungen von der Syntax her sehr ähneln. Zum Schluss folgt in der Diskussion, ob diese Ontologie den Zweck erfüllt, wie und was an ihr bearbeitet werden kann und welche zukünftigen Projekte sich daraus entwickeln können.

5.1. Aufbau

Dieser Abschnitt beschreibt den Aufbau der Ontologie und deren Funktion in Form eines Entity-Relationship-Modells. Zuerst wird der Grundaufbau erläutert, um einen groben Überblick über die Ontologie zu liefern. Die weiteren Abschnitte beschreiben anschließend detailliert die Grundzweige der Ontologie, die aus dem Spezifikationszweig, dem Evaluationszweig und dem Simulationszweig bestehen.

5.1.1. Grundaufbau

Der Grundaufbau zeigt die drei Grundzweige der Ontologie. Das Modell wird in Form eines Entity-Relationship-Modells dargestellt. Subklassen werden mithilfe der *is-a* Beziehung gekennzeichnet. Veranschaulicht wird ein Modell, bei welchen es sich um drei Grundzweigen um Subklassen der Klasse *CAE-Projekt* handelt. CAE-Projekt hat dabei keine Attribute und dient lediglich als zusammenfassende Klasse.

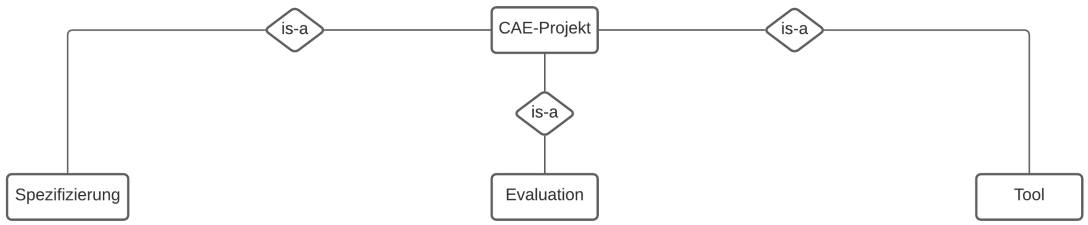

Abbildung 5.1.: Grundaufbau der Ontologie

Wie in Abbildung 5.1 zu sehen ist, bestehen die Grundzweige der Ontologie aus der *Spezifizierung*, der *Evaluation* und den *Tools*. Die Spezifizierung ist dabei ein kleiner Teil der Ontologie. Sie beinhaltet die Information des Auftraggebers und dessen Metadaten. Diese Information bestehen aus Punkten wie Anforderungen an das Produkt und dem sonstigen Schriftverkehr zwischen Auftraggeber und Auftragnehmer. Der Evaluationszweig beinhaltet sämtliche Evaluationsdaten zur Herstellung und zum Testen des Produkts. Diese Daten werden durch das Tool und dessen Analyse- und Simulationsprogrammen zur Verfügung gestellt. Jede Beziehung hat eine korrespondierende inverse Beziehung, solange es keine *is-a* Beziehung ist. In den Modellen sind sie aus Gründen der Übersicht nur als durchsichtige Pfeile gekennzeichnet. Die inverse Beziehung von *hatKontakt* ist beispielsweise *beschreibtKontaktIn*. Jede Klasse A ist generell disjoint mit Klasse B, falls eine nicht die Subklasse der jeweils anderen ist, oder diese keine Beziehung miteinander haben.

5.1.2. Spezifizierung

Die Klasse Spezifizierung beinhaltet die Auftragsanforderungen der Firma. Jede Instanz der Spezifizierung hat ein *Format*.Dies kann beispielsweise eine PDF sein. Dabei beinhaltet jeder Auftrag die Kontaktdaten der Firma, die aus der *PLZ*. *Stadt*. *Straße*, *Hausnummer*, *Telefon*, *Email* und *Fax* bestehen. Die wichtigen direkten Attribute der Spezifizierung sind zu einem die *Anforderungen*

und zum anderen der *Firmenname*. Die Anforderungen umfassen alle Angaben und Bedingungen des Auftraggebers bezüglich des Produkts. Diese sind der Hauptinformationsanteil einer solchen Spezifizierung. Da diese in Form von Stichwörtern oder Fließtext beschrieben werden, wird dafür keine weitere Klasse benötigt, die die Struktur einer solchen Anforderung beschreibt. Stattdessen kann man alles (oder verkürzt) als String einspeichern.

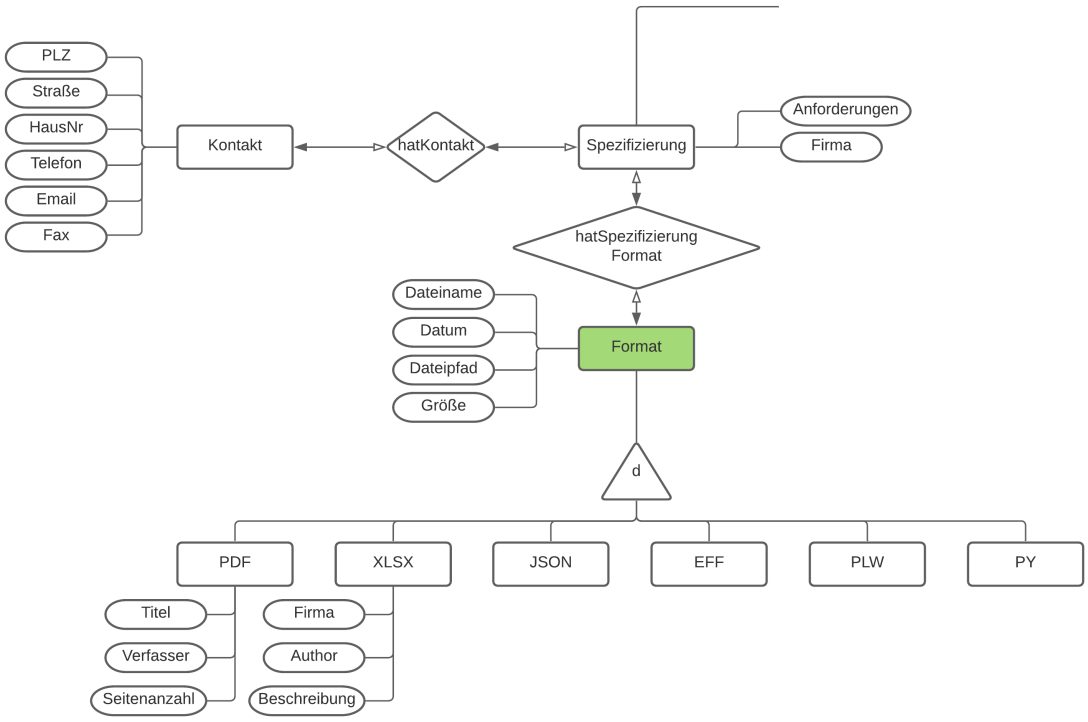

Abbildung 5.2.: Spezifikationszweig der Ontologie

Abbildung 5.2 zeigt dabei den Zweig der Spezifizierung. Die Klasse *Format* ist dabei eine eigenständige Klasse und keine Subklasse von Spezifizierung. Jedes Dokument, unabhängig vom Format, hat dabei die Eigenschaften *Dateiname*, *Datum* der Erstellung, den *Dateipfad*, in dem es gespeichert ist und die *Größe* des Dokuments in kB. Die Klasse Format hat wiederum Subklassen, die das genaue Format beschreiben. Jede Beziehung zu Format hat dabei das Pattern *hatXFormat*, wobei das X ein Platzhalter für die Domain-Klasse ist. Dies hat den Hintergrund, dass jede Beziehung, falls sie verschiedene Klassen beinhaltet und diese nicht identisch sein sollen, gängigerweise auch eigene Namen haben. Die grüne Hinterlegung der Klasse, hat den Hintergrund, dass sie nicht vollständig und somit erweiterbar ist. Es gibt sehr viele Formate, vor allem in der CAE-Domäne. Jedes Tool speichert beispielsweise seine Modelle in ein individuelles Format ab. ANSYS, ein kommerzielles bekanntes Simulationstool, speichert seine Projekte zum Beispiel als .wbdp (ANSYS Workbench Project Global) ab. Jedes dieser Formate in die Ontologie miteinzubeziehen, ist utopisch. So kann bei einem unbekannten Format nur eine Instanz der Klasse Format erstellen anstatt einer genaueren Subklasse. Die Subklassen von Format sind dabei disjunkt und somit kann keine Instanz einer Subklasse gleichzeitig eine Instanz einer anderen sein.

5.1.3. Evaluation

Der Evaluationszweig beschäftigt sich mit den anfallenden Simulationsdaten. Der Großteil dieser heterogenen Daten sind dabei Rohdaten aus der Vielzahl an durchgeführten Simulationen. Jedoch werden diese Rohdaten nicht an die Kunden weitergeleitet, da diese für sie keine große Relevanz haben. So liegt eine Unterscheidung von wichtigen *Entwicklerinformationen* und wichtigen *Kundeninformationen* vor. In diesem Zweig werden diese Informationen und deren Aufbau / Beziehungen beschrieben. Abbildung 5.3 zeigt den Zweig im Modell. Dieser steht in Verbindung mit dem dritten Zweig, der die Simulation in der Domäne beschreiben soll. Klassen des Simulationszweiges sind *Tool* und *CAE-Modell*. Das Simulationstool simuliert mithilfe der CAE-Modelle und der im Tool integrierten Algorithmen und speichert diese Simulationen. Die Klasse *Simulation* ist eine Subklasse von Entwicklerinformation und erbt somit alle Attribute und Beziehungen zu anderen Klassen. So hat auch jede Simulationsinstanz ein Format, da sich diese Simulationen auch abspeichern lassen. Welches Format genau, ist wiederum abhängig vom benutzten Tool.

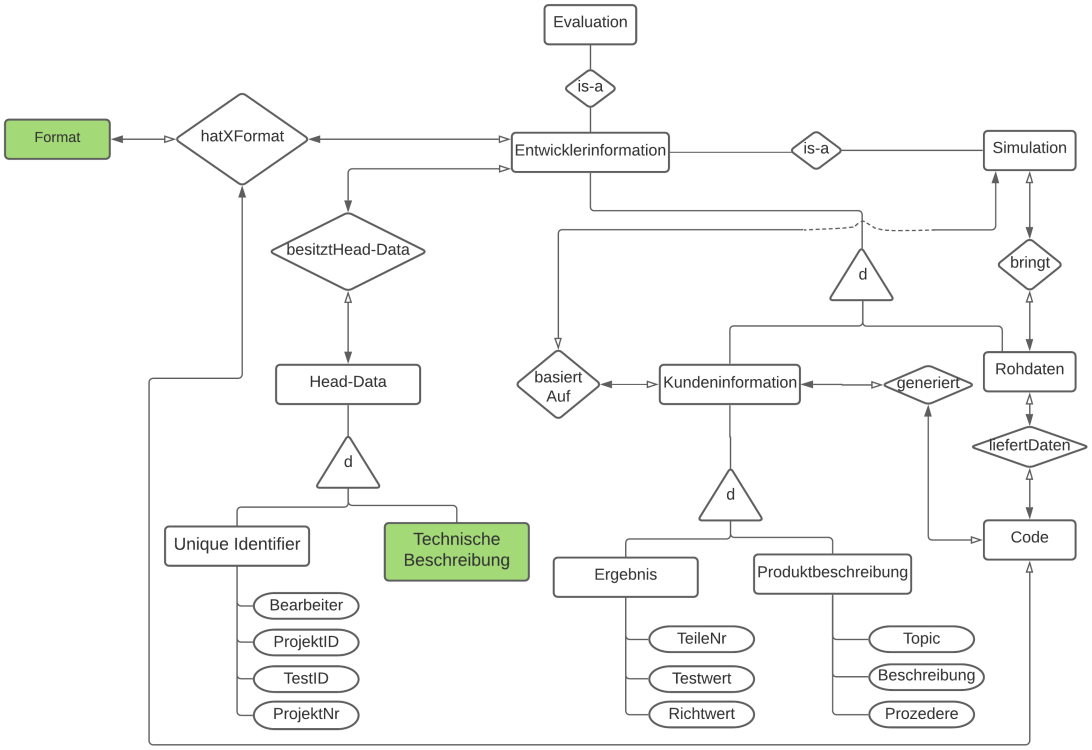

Abbildung 5.3.: Evaluationszweig der Ontologie

Die Simulationen liefern anschließend die besagten Rohdaten und bilden die Grundlage für *Kundeninformation*. Im Anhang A.1 und A.2 befinden sich Beispiele von lesbaren Rohdaten. Welches Format diese Rohdaten haben, hängt von der Simulation und vom Tool ab. So hat A.1 das Format EFF (EverQuest Effects File) und A.2 das Format PRM (Parameter File). Zwar sind diese Rohdaten in dieser Form unübersichtlich und irrelevant für den Kunden, jedoch wird daraus eine Zusammenfassung generiert. Dies kann entweder durch Code oder manuell zu Stande kommen. Diese Zusammenfassung ist dann eine Instanz der *Kundeninformation*.

Die Klasse *Rohdaten* und *Kundeninformation* sind dabei beide Subklassen von *Entwicklerinformation*, jedoch untereinander disjunkt. Jede Zusammenfassung der Rohdaten ist wichtige Kundeninformation und somit auch wichtig für den Entwickler. Neben dem Format einer solchen Zusammenfassung wird sie durch einen *Header* strukturiert, der wichtige Kerninformationen zum Projekt und dem Test enthält. Dies wird durch die Aussage *Entwicklerinformation - besitztHead-Dada - Head-Data* im Modell symbolisiert.
Hier wird die Klasse nochmal unterteilt in *Unique Identifier* und *Technische Beschreibung*. Unique Identifier sind Daten, die in jeder Datei vorhanden sein müssen, um die Zugehörigkeit der Datei zu identifizieren. Attribute wie *TestID*, *ProjektID*, *ProjektNr* und zuständiger *Bearbeiter* sind immer enthalten. Lediglich die Benennung der Attribute, der Datentyp oder das Pattern der Benennung kann sich firmenextern unterscheiden. Jede Firma hat eine gewisse Guideline zur Benennung und Strukturierung von Daten. Daraus generisch eine Ontologie zu entwickeln, die gleichzeitig so detailliert wie möglich ist, gestaltet sich als schwierig. Der Code C.1 im Anhang zeigt ein solches Beispiel, indem die Benennung der Daten und der Dateien nach bestimmten Regex Patterns definiert ist. Regex steht abkürzend für Regulärer Ausdruck und definiert in der theoretischen Informatik Syntaxregeln einer Zeichenkette. Beispielsweise wird festgelegt, dass ein Excel Test Report folgendermaßen aufgebaut ist:

„Testplan- “+ Fünf Nummern + „-TestOrder-“+ Hexazahlen mit Bindestrichen getrennt + „-File-“+ willkürlich vom User gegebener Name. Ein Beispiel wäre also:
TestPlan-36277-TestOrder-f9e1c880-7d44-11e5-ac65-8cdcd418df88-File-36277.

Die *technische Beschreibung* umfasst alle Attribute des zu testenden Objekts. Auch hier ist die Klasse grün hinterlegt, da sie nicht vollständig ist. Die technische Beschreibung eines Produkts ist nicht nur firmenabhängig, sondern auch sehr kontextabhängig. Betrachtet man Abbildung B.1 im Anhang, so lassen sich Beschreibungen wie *Anstromfläche, Filterfläche, Filtertyp,* etc. beobachten. Hier wurde ein Filter getestet und hat dementsprechend die beschreibenden Attribute dafür. Testet man zum Beispiel nun einen Motor, wird er durch komplett andere Attribute wie *Anzahl der Zylinder* oder ähnliches beschrieben. Aufgrund dieser Variabilität muss die technische Beschreibung immer dem Kontext angepasst werden und infolgedessen ist sie hier nur als Klasse ohne Attribute gekennzeichnet. Eine Idee für die Lösung des Problems, wird in Abschnitt 5.1.3.1 erklärt. Nach dem Header folgt anschließend optional eine *Produktbeschreibung*. Diese ist unterteilt in *Topics*, *Beschreibung*, *Prozedere*. Dabei sind die Topics die Gliederung der zu erklärenden Themen, die Beschreibung ist eine Erklärung der Aufgabe und wiederholt sozusagen die Anforderungen der Auftraggeber und das Prozedere ist die Erläuterung, wie die Lösung des Problems erstellt wurde. Darin sind zum Beispiel die Modelle enthalten, deren Simulationen und Graphen zu bestimmten Testergebnissen. So eine Art Produktbeschreibung lässt sich jedoch nicht überall finden. Viele Testergebnisse beinhalten nur den Header, die Testergebnisse an sich und möglicherweise Graphen zur Visualisierung ohne eine textuelle Erklärung dazu (s. Abbildung B.1 im Anhang).
Mit den *Ergebnissen* ist das ähnlich wie mit der technischen Beschreibung. Hier ist immer die Frage des Kontextes wichtig. Im Exemplar B.1 wird ein Filter getestet und somit sind die Ergebnisse beispielsweise genauer mit *Diff -druck* dokumentiert. Um auf das Beispiel mit dem Motor zurückzugreifen, würde hier der Differentialdruck wahrscheinlich nicht vorkommen. Attribute wie *PS* oder *Spritverbrauch* würden da eher anfallen. Demzufolge ist es auch hierbei schwierig, generische Metadaten zu extrahieren. Was jedoch jede Evaluation hat, ist ein Richtwert, der vom Auftraggeber erwartet wird und ein tatsächlicher Testwert, der aus den Simulationen entnommen wird. Was für

eine Einheit diese Werte haben, ist kontextabhängig. Jedoch lässt sich verallgemeinern, dass es sich hierbei um einen Wert des Datentyps *double* handelt. Die *TeileNr* ist dabei die firmeninterne Serienmaterialnummer eines Mediums. Zwar kann sie in verschiedenen Formen vorliegen (z.B. nur Zahlen oder ein anderes Pattern), wird jedoch in jedem Testergebnis beschrieben.

5.1.3.1. Idee der Middleware für die technische Beschreibung

Eine Idee, technische Beschreibungen durch Metadaten zusammenzufassen, ist die Einspeicherung jeglicher Information in einem String. Dadurch verlieren die Daten jedoch an Form. So war eine Zahl in diesem String vorher ein Wert und nun ist sie nur ein Teil des Textes. Auf dieser Grundlage können wenig Informationen bezogen werden. So ist ein weiter Vorschlag, dass der Text mit Information versehen werden kann. Informationen, die beschreiben, ob der nachfolgende Teil eine Zahl, ein Name oder eine weitere beliebige Information ist. Aus dieser Schlussfolgerung, kam der Gedanke jegliche technische Beschreibung als XML-Datei zu beschreiben. Um dies zu realisieren, benötigt es eine Art von Middleware oder Code, das aus Rohdaten beispielsweise gezielt die technischen Informationen herausfiltert und in eine XML-Datei schreibt und abspeichert.
Abbildung 5.4 ist dabei das Konzept der Mittdleware. Die Idee ist, Rohdaten einer Simulation abzufangen, daraus die technischen Daten abzulesen und eine XML-Datei zu konstruieren. Um dies zu ermöglichen, müssten Regeln oder Constraints festgelegt werden. So wäre es notwendig, der Middleware gewisse Keyelements zu defnieren. Diese sollen zeigen, welche Strings innerhalb der Rohdaten die Namen der technischen Beschreibungen sind. Abgebildet im Modell sind das die *Bezeichnungen*. So kann der Bezeichner möglicherweise *Filter* oder ähnliches sein. Wie daraus

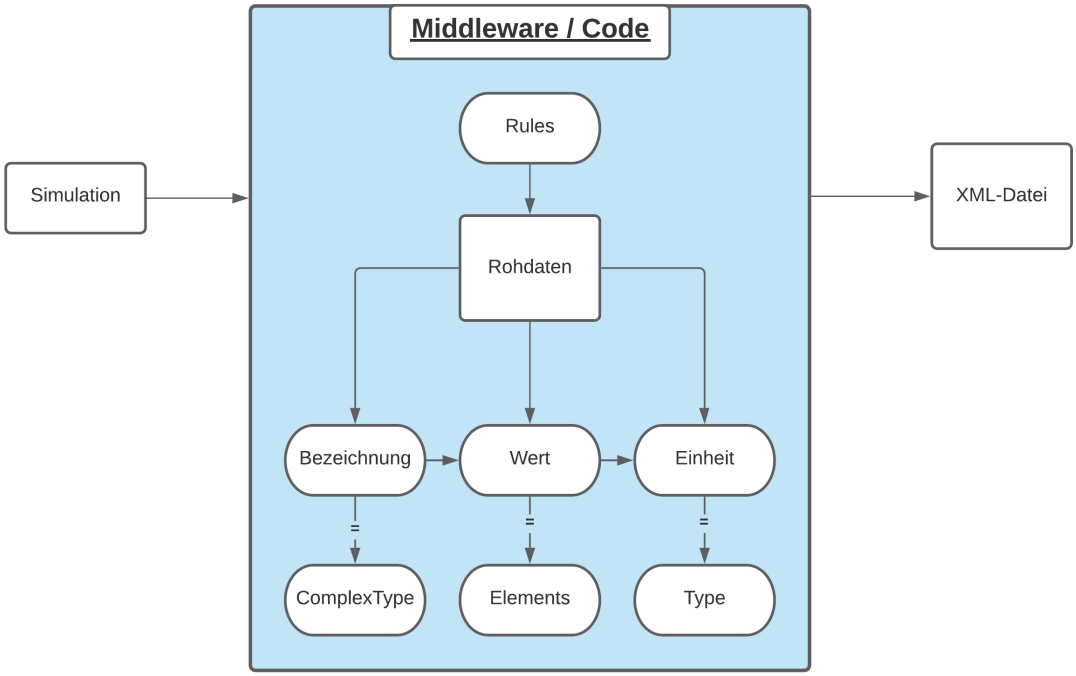

Abbildung 5.4.: Vorschlag zur Strukturierung technischer Beschreibungen

dann die Werte folgen, ist abhängig vom Aufbau der Rohdaten. So könnten die Keyelements und die Werte mit einem „:“ getrennt sein oder in Form einer Tabelle oder sonst wie. Letztendlich besitzt jede Beschreibung ggf. auch eine Einheit. Da in XML Datentypen selber definiert werden können, ist es möglich, eine Bibliothek mit den in der Firma benutzten Datentypen zu besitzen. Bezeichner würden im XML-Dokument dem *ComplexType* gleichen und die Attribute und dessen Werte wären dessen *Elements*. Falls ein Filter durch mehr als nur einem Attribut beschrieben wird, können diese einfach im <*xs:sequence*> aufgelistet werden. Zusätzlich müsste auch manuell eingegeben werden, um welche Datentyp es sich handelt. Das würde der *Einheit* entsprechen und im XML-Dokument dem Type. Handelt es sich beispielsweise um die Temperatur, so ist der Datentyp ein *double*. So könnten technische Daten beschrieben werden, indem ihre Struktur in XML beschrieben wird. Der Nachteil jedoch ist, dass die Regeln und die Keyelements definiert werden müssen. Ob sich das für eine Firma aufgrund einer Ontologie lohnt, ist fragwürdig. Die Idee wird hier dennoch erwogen. Da es sich aber auch nur um einen Vorschlag handelt und nicht der Realität enstpricht, bleibt die Klasse *Technische Beschreibung* abstrakt ohne Attribute.

5.1.4. Simulation

Der Simulationszweig der Ontologie beschäftigt sich mit der Struktur der für die Simulation benötigten Daten. Für jede Simulation benötigt man grundlegend das passende *Tool*. Wie bereits in Abbildung 3.4 gibt es in der CAE-Domäne verschiedene Simulationsarten. Jedes dieser Tools ist dabei fähig eine gewisse Anzahl an *Analysis Types* zu beherrschen und somit die Modelle mit den passenden Algorithmen zu simulieren. Die Hierarchie von *Analysis Types* ist verkürzt aus [al20a] entnommen. Abbildung 5.5 zeigt die detailliertere Hierarchie, wobei hier aufgrund der Relevanz und der Länge das Hauptmerkmal nur auf die Haupttypen gelegt wurde. Jedes dieser Simulationsmethoden aus Abbildung 3.4 lässt sich in eines dieser Subktypen einordnen. Die folgenden Definitionen erläutern die jeweiligen Subtypen.

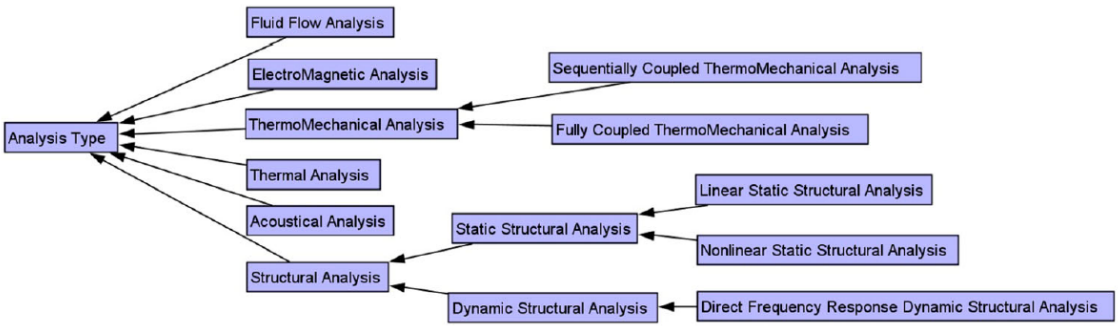

Abbildung 5.5.: Analysis Types Hierarchie (übernommen aus [al20a])

Definition 5.1 (Numerische Strömungsmechanik)
Die numerische Strömungsmechanik (englisch Computational Fluid Dynamics, CFD) ist eine etablierte Methode der Strömungsmechanik. Sie hat das Ziel, strömungsmechanische Probleme approximativ mit numerischen Methoden zu lösen. Die benutzten Modellgleichungen sind meist Navier-Stokes-Gleichungen, Euler-Gleichungen, Stokes-Gleichungen oder Potentialgleichungen [phy09].

Definition 5.2 (Elektromagnetische Analyse)
Die Elektromagnetische Analyse testet die elektromagnetische Verträglichkeit (EMV) eines Modells. Dabei wird die elektromagnetische Verträglichkeit typischerweise definiert als die Fähigkeit eines Apparats, einer Anlage oder eines Systems, in der elektromagnetischen Umwelt zufriedenstellend zu arbeiten, ohne dabei selbst elektromagnetische Störungen zu verursachen, die für alle in dieser Umwelt vorhandenen Apparate, Anlagen oder Systeme unannehmbar wären [Mag13].

Definition 5.3 (Thermomeachanische Analyse)
Eine Technik, bei der eine Verformung der Probe unter nicht oszillierender (nicht schwingender) Spannung gegen Zeit oder Temperatur überwacht wird, während die Temperatur der Probe in einer bestimmten Atmosphäre programmiert ist. Die wirkende Kraft kann Kompression, Spannung, Biegung oder Torsion sein [hit].

Definition 5.4 (Thermale Analyse)
Ist eine Gruppierung von Techniken, die die physikalische und chemische Eigenschaften eines Materials unter Spannung von Termperatur untersucht. Die Thermomechanische Analyse ist dabei eine Teilmenge davon.

Definition 5.5 (Akustische Analyse)
Akustische oder Schallanalyse ist die Messung von Schallwellen, die durch Kontakte von Komponenten innerhalb eines Produkts verursacht werden [Mac].

Definition 5.6 (Strukturelle Analyse)
Bei der Strukturellen Analyse werden, im Zusammenhang der CAE-Domäne, die Auswirkungen von Lasten auf einer Struktur oder einem Objekt bestimmt.

Abbildung 5.6 zeigt den kompletten Simulationszweig. Neben den Analysetypen besitzt jedes Tool die Möglichkeit in der Simulation, äußere Kräfte oder sonstige Faktoren auf ein Modell wirken zu lassen. Dies wird im Modell durch die Klasse *External Loads* symbolisiert. Diese External Loads korrespondieren dabei mit dem Analysetypen. Basiert die Simulation beispielsweise auf einer Thermomechanischen Analyse, so können Temperatur oder Druck auf ein Modell von außen ausgeübt werden. Die Analyse besteht dann darin, die Veränderung des Modells auf diese äußeren Kräfte zu beobachten. So befindet im Anhang D.1 ein Bild mit dem Beispiel von SolidWorks und dessen External Loads. Während der Simulation wird das Modell in Flächen, sog. Faces eingeteilt. Diese External Loads können anschließend einem oder mehreren Faces zugeordnet werden und somit die gewünschte Kraft an der Stelle ausüben. Trotzdem ist diese Klasse hier in dem Modell ohne Attribute, da diese External Loads auch sehr Tool abhängig sind. SolidWorks simuliert zum Beispiel nur auf Basis der Finite-Element-Method und besitzt somit die dementsprechenden Kräfte. Ein anderes Tool würde möglicherweise andere Kräfte besitzen.

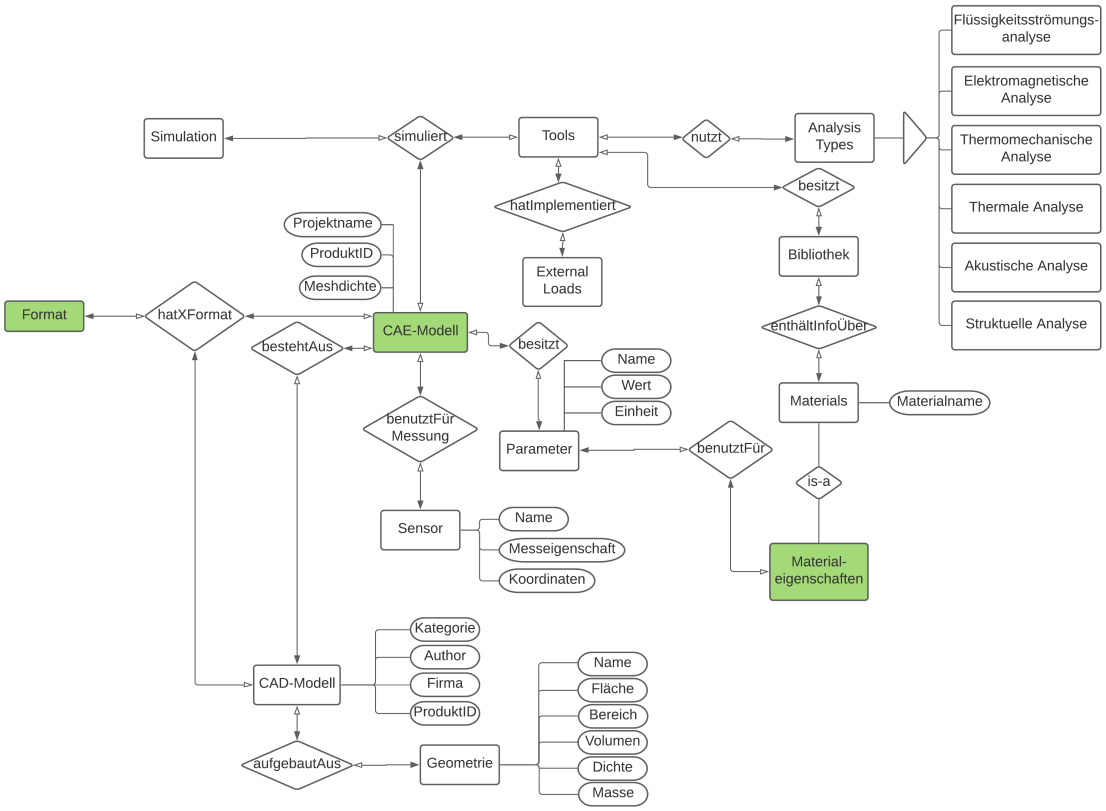

Abbildung 5.6.: Simulatioszweig der Ontologie

Jedes Tool hat jedoch eine *Bibliothek*, die Information wie zum Beispiel Simulationsalgorithmen oder wichtige Informationen bzgl. des benutzten *Materials*. Jedes Modell ist aus einem oder mehreren Materialien aufgebaut, das verschiedene Materialeigenschaften besitzt. Dabei hat die Klasse *Materials* das Attribut Materialname und die erbende Subklasse *Materialeigenschaften* beschreibt die Werkstoffeigenschaften des jeweiligen Materials. Aus Gründen der Strukturierung wurden die Materialeigenschaften nochmal eingeteilt in *Mechanische Eigenschaften*, *Physikalische Eigenschaften* und *Chemische Eigenschaften* (s. Abbildung 5.7). Diese Subklassen sind jedoch nicht disjunkt, da Attribute wie zum Beispiel *Temperatur* wiederholt vorkommen können. Wären sie disjunkt, dürften sie sich keine Attribute teilen und man müsste ggf. die Temperaturvariablen leicht umbenennen, was jedoch sinnlos ist. Die genauen Metadaten wurden aus den Beispielmodellen aus SolidWorks und Beispielprojekten von ANSYS extrahiert.

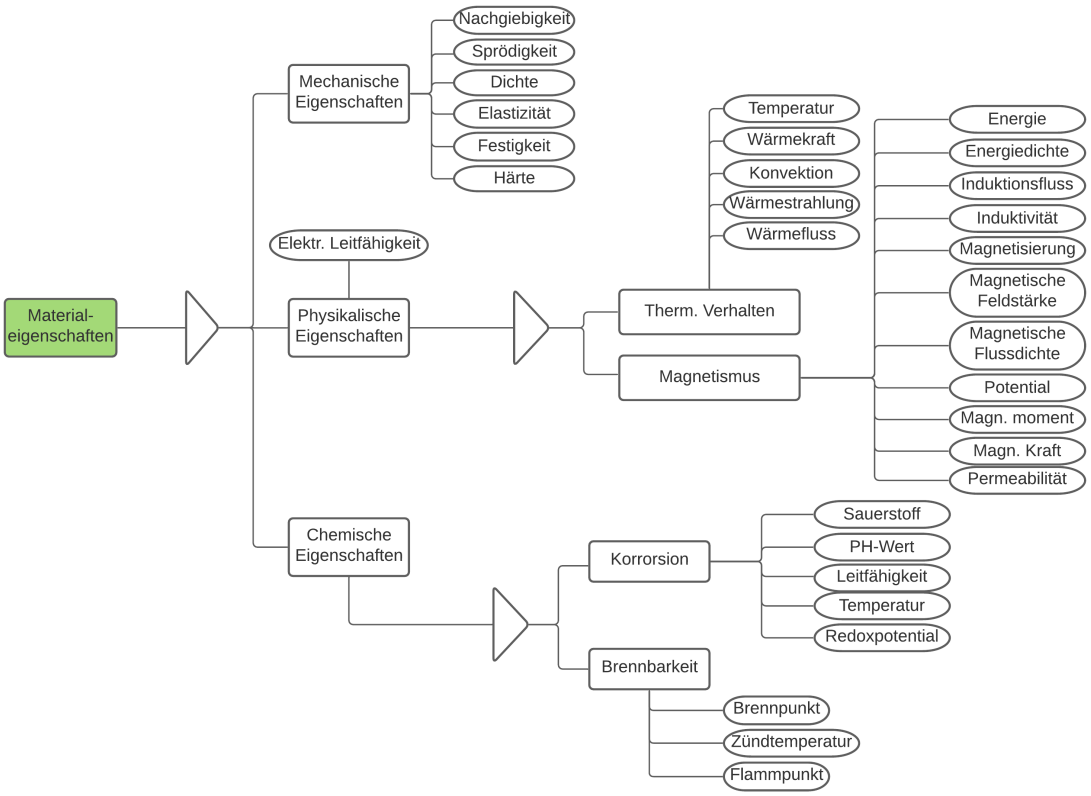

Abbildung 5.7.: Materialeigenschaften in der Ontologie

So ist der Code 5.1 ein kleiner Ausschnitt eines ANSYS Projekts. Zu sehen ist die Beschreibung von verschiedenen Parametern im Modell. Die Zahl 379 in Zeile 24 nennt die Nummer des Parameters und somit auch die Tatsache, dass hier ein Modell 379 verschiedene Parameter besitzt. Neben der Zahl, werden diese Parameter mit einem Namen und der dazugehörigen Einheit beschrieben. Beispielsweise hat Parameter 375 die Bezeichnung *Volumetrischer Durchfluss* und wird mit der Einheit $\frac{m^3}{s}$ beschrieben. Bei jedem dieser Parameter besteht die Möglichkeit, mithilfe des Booleans *isSupressed*, diesen zu unterdrücken, falls er nicht benötigt wird. In der Ontologie sind jedoch deutlich weniger vorhanden. Dies hat den Hintergrund, dass eine solche Anzahl an Parametern viel zu unübersichtlich wäre. Zumal viele sehr speziell, oft für die Simulation irrelevant, nicht veränderbar sind und/oder keinen Einfluss auf die Materialeigenschaften haben. Viele Parameter lassen sich auch durch gewisse Daten ableiten. Betrachtet man ein Leitungsrohr und kennt dessen Strecke und Radius, ist die Berechnung des Volumetrischen Durchflusses simpel. Dieser beeinflusst auch nicht das Modell, sondern dient lediglich zur Beschreibung des Modells und wird seitens des Tools gespeichert (nicht im Modell an sich). Aus diesem Grund gleichen sich die Klassen *Parameter* und *Materialeigenschaften* nicht. So ist die Gravitation vergleichsweise ein wichtiger Parameter (v.a. für sog. Falltests), jedoch immer konstant und keine Materialeigenschaft. In der Ontologie sind dabei die wichtigsten materialbeschreibenden Parameter aus der Werkstoffkunde enthalten und zur Strukturierung in die jeweiligen Bereiche eingeteilt. Nichtsdestotrotz ist die Klasse *Materialeigenschaften* für die Erweiterung markiert, falls nötig. Materialeigenschaften

können entweder extern als bereits vorhandene Datenbank herunterladen oder individuell selber mit den gewünschten Parametern im Tool erstellt werden.

```
.
.
.
<Object Name="/Units/Quantity:Quantity␣375" Version="1.0">
        <class-type valType="String">Quantity</class-type>
        <object-name valType="String">Quantity 375</object-name>
        <member-data valType="String">{"DisplayText": "Volumetric␣Flow␣in", "QuantityName"
: "Volumetric␣Flow␣in", "Unit": "m^3␣s^-1", "IsSuppressed": False }</member-data>
      </Object>
      <Object Name="/Units/Quantity:Quantity␣376" Version="1.0">
        <class-type valType="String">Quantity</class-type>
        <object-name valType="String">Quantity 376</object-name>
        <member-data valType="String">{"DisplayText": "Warping␣Factor", "QuantityName": "
Warping␣Factor", "Unit": "m^6", "IsSuppressed": False }</member-data>
      </Object>
      <Object Name="/Units/Quantity:Quantity␣377" Version="1.0">
        <class-type valType="String">Quantity</class-type>
        <object-name valType="String">Quantity 377</object-name>
        <member-data valType="String">{"DisplayText": "Surface␣Power␣Density", "
QuantityName": "Surface␣Power␣Density", "Unit": "W␣m^-2", "IsSuppressed": False }</member-
data>
      </Object>
      <Object Name="/Units/Quantity:Quantity␣378" Version="1.0">
        <class-type valType="String">Quantity</class-type>
        <object-name valType="String">Quantity 378</object-name>
        <member-data valType="String">{"DisplayText": "Force␣Density", "QuantityName": "
Force␣Density", "Unit": "N␣m^-3", "IsSuppressed": False }</member-data>
      </Object>
      <Object Name="/Units/Quantity:Quantity␣379" Version="1.0">
        <class-type valType="String">Quantity</class-type>
        <object-name valType="String">Quantity 379</object-name>
        <member-data valType="String">{"DisplayText": "Surface␣Force␣Density", "
QuantityName": "Surface␣Force␣Density", "Unit": "N␣m^-2", "IsSuppressed": False }</member-
data>
      </Object>
    </Container>
  </Containers>
</Storage>
```

Listing 5.1: Parameter im ANSYS Modell

So zeigt Abbildung 5.8 ein Beispiel. Hierbei wurde *Strctural Steel* als Material importiert und *Aluminuim* selber mit den gewünschten Parametern erstellt. Jedes dieser Parameter hat eine vorgegebene Einheit.
Abhängig davon sind die Parameter des Modells. Besteht ein Modell aus Aluminium hat es andere physikalische, chemische und mechanische Eigenschaften als Plastik. Jedes dieser Matrialien reagiert auch dementsprechend anders auf äußere, veränderbare Einflüsse wie Temperatur oder Stress. Ein *CAE-Modell* besteht dabei aus ein oder mehreren *CAD-Modellen*. Die wichtigsten Metadaten sind bei beiden Klassen ungefähr gleich. Ein Unterschied liegt im Aufbau der Modelle. Während CAD-Modelle nur Zusammenstellung mehrerer Geometrien sind und somit eher nur eine simple Vektorgrafik ist mit wenig beschreibenden Parametern wie Dichte und Dicke, bestehen

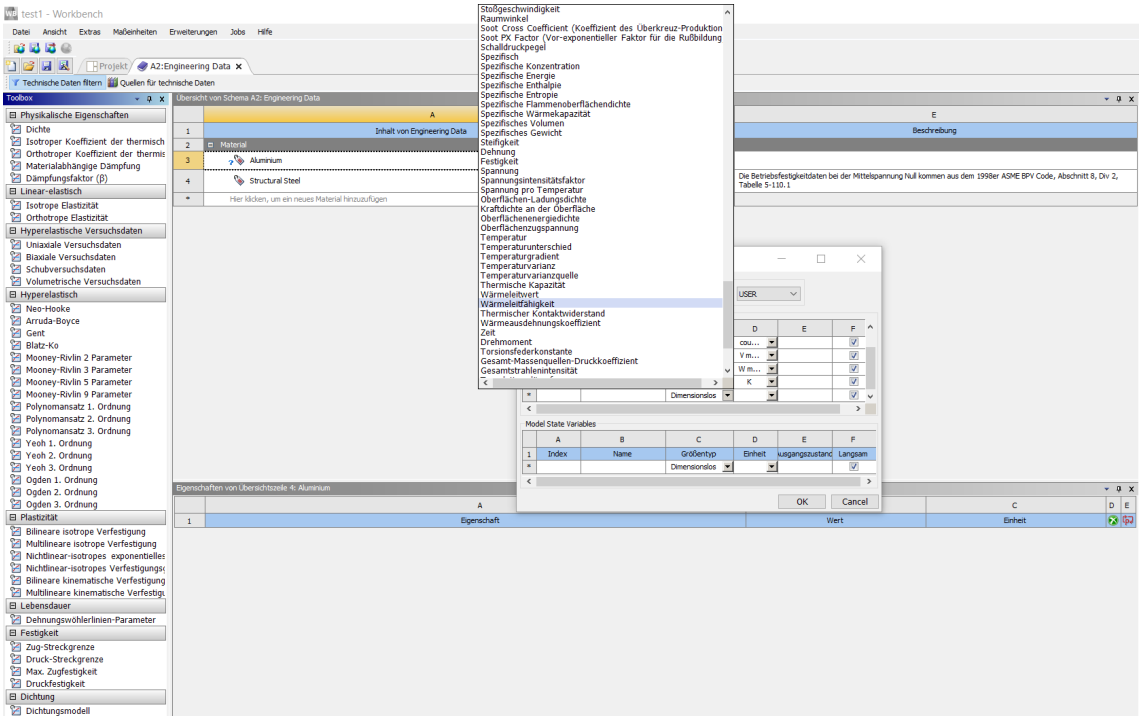

Abbildung 5.8.: Parameter in ANSYS

CAE-Modelle aus Polygonen. So zeigt Abbildung 5.9 einen Sternmotor in JT2Go. Die daraus resultierenden Metadaten werden in der unteren Hälfte der Abbildung wiedergegeben. Beide Modellarten haben Attribute, wie den *Namen*, die *ProduktID* bzw. den Names des einzelnen Modells und der dazugehörige *Projektname*. Weitere beschreibende Attribute wie beispielsweise *Author* o.ä., werden in der Klasse *Format* spezifiziert. Ein CAE-Modell hat zusätzlich noch die Meshdichte. Meshing bezeichnet den Vorgang, die Geometrien eines CAD-Modells in ein diskretisiertes Rechengitter aus vielen Polygone (z.B. Prismen, Pyramiden, usw.) umzuwandeln. Die *Meshdichte* ist dementsprechend ein Indiz wie viele Polygonen benutzt werden, um eine Geometrie zu diskretisieren. Wie in Firgur 5.10 zu sehen ist, entspricht die Meshdichte dem Durchmesser des Polygons in einem Kreis.

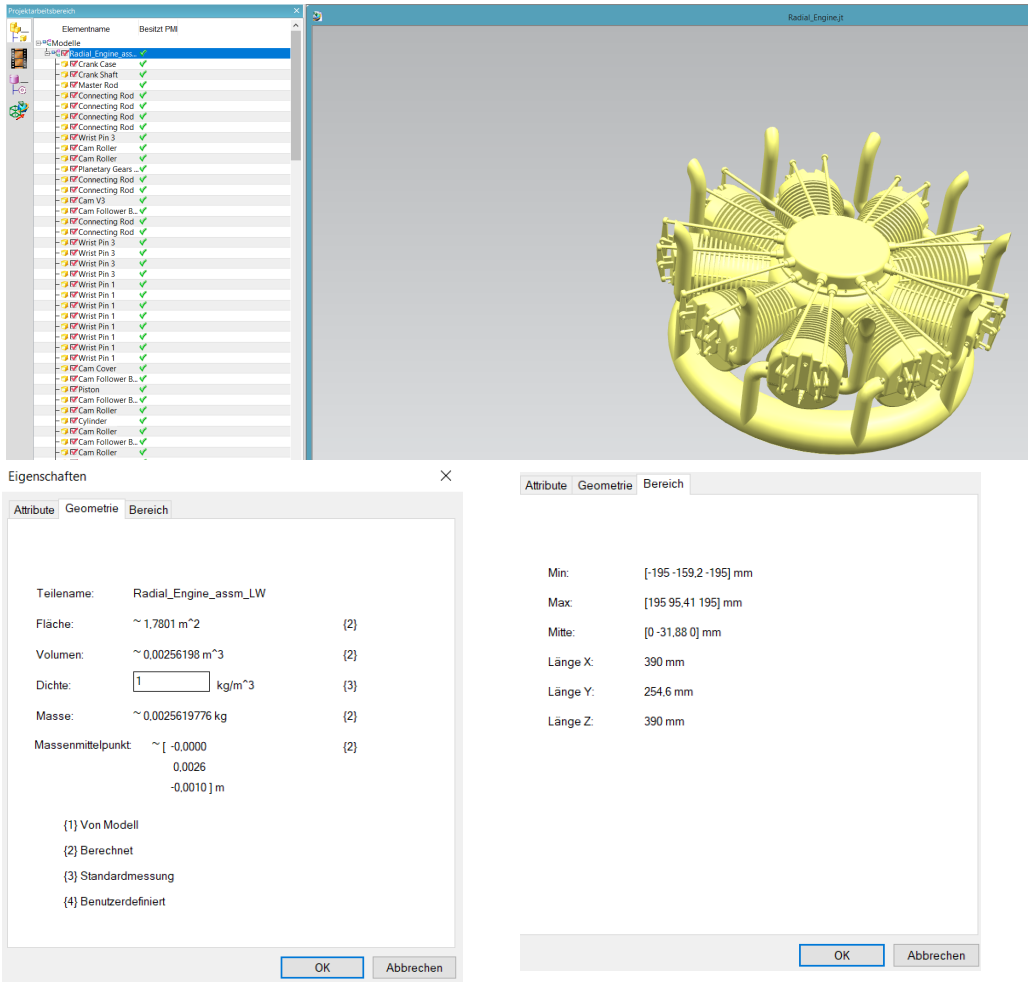

Abbildung 5.9.: Beispiel eines CAD-Modells in JT2Go

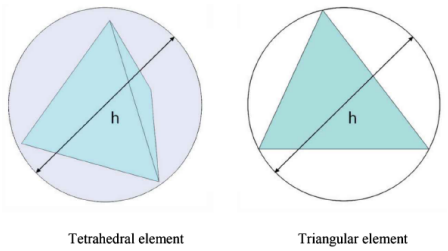

Abbildung 5.10.: Meshdichte eines Polygons (übernommen aus [Pau18])

5.2. OWL-Datei

Die eigentliche Ontologie liest sich in Form einer OWL-Dateil. Diese ist aufgrund der Länge (über 1800 Zeilen ohne Individuen) nicht in der Bachelorarbeit integriert worden, sondern wird extern als OWL-Dokument abgegeben. In der Ontologie werden die jeweiligen Einheiten einer Aussage (vgl. 2.3) einzeln beschrieben und nicht verschachtelt wie im Code 2.1. So ist es

zwar notwendig, bei jedem Attribut und jeder Beziehung zusätzliche Tags hinzuschreiben und zu definieren, welcher Domain (Klasse) sie angehören, jedoch wird daraus folgend die Übersicht und somit auch die Maintainabilty des Codes verbessert, sofern eine Ergänzung notwendig wird.

```
1  <!-- aufgebautAus -->
2
3      < owl:ObjectProperty rdf:about =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie# aufgebautAus ">
4          < rdfs:domain rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#CAD-Modell"/>
5          < rdfs:domain rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#CAE-Modell"/>
6          < rdfs:range rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/ CAE
   -Ontologie#Geometrie"/>
7      </ owl:ObjectProperty >
8  .
9  .
10 .
11  <!-- Author -->
12
13     < owl:DatatypeProperty rdf:about =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#Author">
14         < rdfs:domain rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#CAD-Modell"/>
15         < rdfs:domain rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#CAE-Modell"/>
16         < rdfs:domain rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#Excel"/>
17         < rdfs:domain rdf:resource =" http: // www.semanticweb.org / denis / ontologies /2020/11/
   CAE-Ontologie#PDF"/>
18         < rdfs:range rdf:resource =" http: // www.w3.org /2001/ XMLSchema#string"/>
19     </ owl:DatatypeProperty >
20 .
21 .
22 .
23 <!-- Physikalische_Eigenschaften -->
24
25     < owl:Class rdf:about =" http: // www.semanticweb.org / denis / ontologies /2020/11/ CAE-
   Ontologie# Physikalische_Eigenschaften ">
26         < rdfs:subClassOf rdf:resource =" http: // www.semanticweb.org / denis / ontologies
   /2020/11/ CAE-Ontologie# Materialeigenschaft "/>
27     </ owl:Class >
28 .
29 .
30 .
31 <!-- Disjoint: Format SubClasses -->
32
33     < rdf:Description >
34         < rdf:type rdf:resource =" http: // www.w3.org /2002/07/ owl# AllDisjointClasses "/>
35         <owl:members rdf:parseType =" Collection ">
36             < rdf:Description rdf:about =" http: // www.semanticweb.org / denis / ontologies
   /2020/11/ CAE-Ontologie#DAC"/>
37             < rdf:Description rdf:about =" http: // www.semanticweb.org / denis / ontologies
   /2020/11/ CAE-Ontologie#EFF"/>
38             < rdf:Description rdf:about =" http: // www.semanticweb.org / denis / ontologies
   /2020/11/ CAE-Ontologie#Excel"/>
```

```
39              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#JSON"/>
40              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#PDF"/>
41              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#PLW"/>
42              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#PNG"/>
43              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#PRM"/>
44              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#Py"/>
45              < rdf:Description    rdf:about =" http: // www.semanticweb. org / denis /  ontologies
    /2020/11/ CAE-Ontologie#TFF"/>
46          </owl:members>
47      </ rdf:Description >
```

Listing 5.2: Ausschnitt aus der CAE-Ontologie

Code 5.2 ist dabei ein Ausschnitt aus der Ontologie. So wird dargestellt, wie die Beziehung „aufgebautAus“, das Attribut „Author“ die Klasse „Physikalische Eigenschaft“ und das Axiom der Subklassen von „Format“ kodiert sind. Jede Instanz der Ontologie, sei es jetzt die Klasse, das Attribut oder eine Beziehung, besitzt als eindeutige Indentifikation die in 2.3.2 erwähnten URIs. Die URI der Ontologie lautet *http://www.semanticweb.org/denis/ontologies/2020/11/CAE-Ontologie*. Diese können frei benannt werden. Üblich ist es, die Ontologie online auf einem Webserver zur Verfügung zu stellen und so den Zugriff auf die Ontologie zu ermöglichen. Da die Abgabe dieser Ontologie digital erfolgt, ist die Wahl der URI irrelevant. Aufgrund dessen, dass für die spätere Diskussion *Protégé* zur Konstruktion der Beispieldatenbank benutzt wird, entspricht die URI der URI, die dieser Ontologie standardmäßig von Protégé zugeordnet wurde.
Jeder dieser Abschnitte ist im Kern ähnlich aufgebaut. Die erste Beschreibung bezieht sich aus der Owl-Ontologie und zeigt, um welche Ressource es sich handelt. So werden Klassen mit *owl:Class*, Beziehungen mit *owl:ObjectProperty* und Attribute mit *owl:DatatypeProperty* gekennzeichnet. Die darauf folgende URI ist anschließend der genaue Identifikator der Ressource. Diese besteht aus der URI der Ontologie an sich und nach der # mit dem Namen der spezifischen Ressource. Demzufolge ist *http:// www.semanticweb. org / denis/ ontologies/2020/11/CAE−OntologieAuthor* die URI von *Author*. Attribute und Beziehungen haben zusätzlich die Schilderung, zu welcher *Domain* (Klasse) sie gehören und welchen Wertebereich (*range*) sie besitzen. Der Wertebereich unterscheidet sich, indem das „Ziel“einer Beziehung, eine andere Klasse ist (hier: Geometrie) und das Ziel der Attribute, ein Datentyp (hier: String). Da die Klassen *CAE-Modell* und *CAD-Modell* aus Geometrien aufgebaut sind, sind diese jeweils auch als Domain in der Ontologie gekennzeichnet. Das gleiche gilt für die Klassen *Excel* und *PDF*, die sich mit *CAD-Modell* und *CAE-Modell* das gleiche Attribut (*Author*) teilen.
Klassen haben dieselbe initiale Beschreibung wie Beziehungen und Attribute, nur mit dem Typ *owl:Class*. Alles darauffolgende sind genauere Charakterisierungen der Klassen. So ist die Klasse *Physikalische Eigenschaft* eine Subklasse von *Materialeigenschaften* (*http:// www.semanticweb. org / denis/ ontologies/2020/11/CAE−Ontologie Materialeigenschaf*).
Der letzte Abschnitt kodiert dabei ein Axiom. *Owl:members* beschreibt eine *Collection* von Mitgliedern innerhalb des Abschnitts, die entweder mit dem *owl:AllDifferent*, *owl:AllDisjointClasses* oder *owl:AllDisjointProperties* Axiom erweitert werden. Bei Owl:AllDifferent sind die Klassen paarweise verschieden, jedoch können sie sich einige Individuen teilen. *owl:AllDisjointClasses*

hingegen beschreibt die Eigenschaft, dass Schnitt beider Klassen die leere Menge ist und sie sich somit keinerlei Individuen teilen. Das gleiche gilt mit *owl:AllDisjointProperties* für die Beziehungen und Attribute. Im *owl:members* Container werden dann die richtigen Klassen aufgezählt.

5.3. Diskussion

In der Diskussion wird die Konsistenz und Funktionalität der Ontologie geprüft und notiert, welche Verbesserung sie haben kann und inwiefern Möglichkeiten der Benutzung der Ontologie liegen. Die Korrektheit wird dabei in Protégé geprüft. Protégé stellt für diesen Zweck einen integrierten Reasoner zur Verfügung. Dieser Reasoner ist dafür zuständig, die angegeben Axiome zu überprüfen und ob sie oder die einzelnen Ressourcen der Ontologie im Konflikt miteinander stehen. Kurz erklärt, versuchen Reasoner ein Modell der Ontologie mit den gegebenen Axiomen aufzubauen und überprüfen, ob bei Erstellung der Individuen alle Axiome eingehalten werden können oder ob es Konflikte gibt. Die Korrektheit wird, um genauer zu sein, mithilfe der *consistency* und der *coherence* ausgedrückt [SSL14]. Der Unterschied wird anhand eines Beispiels erklärt.
Sei *Gerade* die Klasse der geraden Zahlen und *Ungerade* die Klasse der ungeraden Zahlen. Diese Klassen würde auch mit *owl:ComplementOf* gekennzeichnet werden, da eine Zahl logischerweise nicht gerade und ungerade gleichzeitig sein kann. Nehmen wir an, wir haben eine dritte Klasse *Zahl* und sagen, dass sie Subklasse von *Gerade* und *Ungerade* gleichzeitig ist. So ist diese Ontologie ohne ein Individum zwar konsistent, da der Reasoner korrekte Modelle aufbauen kann (ohne eine Instanz der Klasse *Zahl*), jedoch ist die Klasse *Zahl* an sich *inkohärent*. Erstellt man nun eine Instanz von *Zah*l, so ist diese Instanz gleichzeitig gerade und ungerade, was durch das Axiom *ComplementOf* verboten wurde. Mit dieser Instanz in der Ontologie ist kein Modell mehr konsistent und somit ist die Ontologie nicht konsistent. In der Situation wird dann versucht, die Ontologie zu überarbeiten. Ein Lösungsvorschlag hierbei wäre , die Klasse *Zahl* als Superklasse zu nehmen und die Klassen *Ungerade* und *Gerade* als dessen Subklassen.
Diese Axiome, die überprüft werden, liegen entweder direkt vor oder werden durch sog. weitere Inferenzen des Reasoners entdeckt. Inferenzen sind Schlussfolgerungen des Reasoners auf Basis der bestehenden Regeln (Axiomen). So kann beispielsweise eine Inferenz aus nur einem Axiom begründet werden oder aus einer Vielzahl an Verkettungen von Axiomen. Der Reasoner ist dabei in der Lage, einfache Inferenzen zu schließen. Wird ein Individuum beispielsweise hinzugefügt, das eine korrekte Beziehung zu einem anderen Individuum besitzt und diese Beziehung eine korrespondierende inverse Beziehung hat, so deklariert der Reasoner die inverse Beziehung automatisch aufgrund des Axioms *inverseOf*. Bei großen Ontologien gibt es potentiell langkettige Inferenzen. Diese sind oft zu komplex für den Reasoner. Die Möglichkeit diese Inferenzen zu benutzen, besteht jedoch trotzdem. So können über gewisse Definitionssprachen wie Semantic Web Rule Language (SWRL) eigene Verkettungen und Regeln definiert werden, die der Reasoner in der Ontologie überprüft und somit auch komplexere Inferenzen ermöglicht.
Importiert man die Ontologie in Protégé und startet den Reasoner, so folgt keine Fehlermeldung. Die Ontologie ist also konsistent. Um zu zeigen, dass der Reasoner jedoch erkennt, wenn Axiome nicht eingehalten werden, wird wissentlich zur Präsentation ein Fehler eingebaut. Dieser Fehler kann von jeglicher Art sein, muss aber der Ontologie widersprechen. So kann beispielsweise einer Instanz eine falsche Beziehung oder ein falsches Attribut zugeordnet werden. In dieser Demonstration wird eine Instanz der Klasse *Spezifikation* erzeugt. Diese wird unter anderem beschrieben von einem Format und einer Kontakt-Instanz. Zusätzlich hat sie die Attribute *Anforderungen* und die zuständige

Firma. Bis zu dem Punkt ist alles korrekt. Nun weisen wir dieser Instanz das Attribut *Meshdichte* und die Beziehung *bestehtAus* und haben als Range eine Instanz der Klasse *CAD-Modell*. Diese Zuweisungen sind offensichtlich falsch und widersprechen der Ontologie. *Meshdichte* ist ein Attribut der Klasse *CAE-Modell* und *bestehtAus*, hat als Domain eine Instanz der Klasse *CAE-Modell*, wobei unser Individuum eine Instanz der Klasse *Spezifikation* ist.
Führen wir nun den Reasoner aus, weist er uns darauf hin, dass unsere Ontologie inkonsistent ist und dieser keine nützlichen Inferenzen aus dieser Ontologie deuten kann. Was Inferenzen genau sind, wird später detailliert erläutert. Zusätzlich liefert der Reasoner uns 18 Erklärungen, wieso diese Aussagen im Konflikt mit der Ontologie stehen. Davon sind manche direkt und manche indirekt hergeleitet. Erklärung 5 wäre zum Beispiel eine offensichtliche Erklärung.

Erklärung 5:

1) *Spezifikation4894* Type **Spezifikation**
2) bestehtAus Domain **CAE-Modell**
3) DisjointClasses: **CAE-Modell**, …
4) *Spezifikation4894* bestehtAus *CAD-Modell4894_1*

So wurde das Individum *Spezifikation4894* manuell als Instanz von Typ *Spezifikation* festgelegt (1). Jedoch hat das Object Property bestehtAus die Domain **CAE-Modell** (2). In (3) überprüft der Reasoner, ob die zwei Klassen möglicherweise identisch sein können. Da aber, wie in 4.1 erwähnt, alle Klassen, die keine Beziehung zueinander haben, als DisjointClasses deklariert werden, wird diese Möglichkeit auch ausgeschlossen. Zuletzt wird in (4) die genaue Aussage erwähnt, die die Ontologie inkonsistent in Bezug auf der Betrachtung der in (1)-(3) erwähnten Axiome macht.
Weitere Erklärungen werden indirekt vom Reasoner als Inferenzen aufgenommen. Eine eher indirekte Inferenz wäre die aus Erklärung 15:

Erklärung 13:

1) hatKontakt Domain **Spezifikation**
2) bestehtAus inverseOf bildet
3) DisjointClasses: **CAE-Modell**, …
4) *Spezifikation4894* hatKontakt *Spezifikation4894_Format*
5) *Spezifikation4894* bestehtAus *CAD-Modell4894_1*
6) bildet Range **CAE-Modell**

In dieser Erklärung nimmt sich der Reasoner erst die Aussage, dass *Spezifikation4894* eine Beziehung enthält mit dem Namen hatKontakt, die die Domain **Spezifikation** hat (1). Die zweite (und falsche) Beziehung aus der Instanz, ist die bestehtAus, die als Axiom eingetragen hat, dass sie das inverse von bildet ist (2). In (3) zeigt der Reasoner wie im vorherigen Beispiel, dass beide Klassen nicht gleich sein können. (4) und (5) belegen, aus welchen Aussagen (1) und (2) abgeleitet werden und (6) stellt anschließend den Widerspruch dar. So hat bildet die Range **CAE-Modell** und das Individum *CAD-Modell4894_1* bildet eine Aussage durch die Beziehung mit einem Individum des Typs **Spezifikation**. Weitere Erklärungen folgen beispielsweise aus dem Attribut *Meshdichte*, das als Domäne CAE-Modell angegeben hat. Die Klassen *Spezifikation* und *CAE-Modell* sind jedoch

disjoint und können sich aus diesem Grund nicht gleichen.
So ist es, wie bei den herkömmlichen relationalen Datenbanken, auch hier möglich, mithilfe von Datenbanksprachen die strukturierten Daten zu verwalten. Während in relationalen Datenbanken beispielsweise SQL diese Rolle übernimmt, benötigt man für die RDF-Struktur eine angepasste Sprache. SPARQL Protocol and RDF Query Language (SPARQL) ist bei RDF-Strukturen die Standardsprache. SPARQL wurde entwickelt, um verknüpfte Daten für das Semantic Web zu ermöglichen. Ziel ist es, Daten durch Verknüpfung mit anderen globalen semantischen Ressourcen anzureichern und so Daten auf sinnvollere Weise zu teilen, zusammenzuführen und wiederzuverwenden. Infolgedessen kann die Leistungsfähigkeit von SPARQL zusammen mit der Flexibilität von RDF die Entwicklungskosten senken, indem das Zusammenführen von Ergebnissen aus mehreren Datenquellen vereinfacht wird [Ont].
So wie jede Aussage in OWL, besteht eine Aussage in SPARQL auch aus einem Subjekt, einem Prädikat und einem Objekt. Da Protégé ein SPARQL Plugin zur Verfügung stellt, folgt eine Beispiel-Abfrage zur CAE-Ontologie in Code 5.3.

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX owl: <http://www.w3.org/2002/07/owl#>
3 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
4 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5 PREFIX cae: <http://www.semanticweb.org/denis/ontologies/2020/11/CAE-Ontologie#>
6
7 SELECT *
8    WHERE { ?Spezifikation a cae:Spezifikation.
9            ?Spezifikation cae:hatKontakt ?Kontakt.
10           ?Spezifikation cae:hatSpezifizierungFormat ?Format.
11 }
```

Listing 5.3: SPARQL query

Der Header, bestehend aus den Prefixen, importiert die zu benutzenden Ontologien. Die ersten vier sind immer dabei und ermöglichen die Strutktur einer Ontologie. In Zeile 5 ist die CAE-Ontologie importiert und wird durch *cae* abgekürzt. Die folgende Queuery liefert alle Individuen der Klasse *Spezifikation* und deren Beziehungen zu den korrespondierenden Individuen der Klasse *Kontakt* und *Format* in Form der Tabelle 5.1.

Aus diesen lassen sich auch die Attribute oder weiterführende Beziehungen zu anderen

Spezifikation	Kontakt	Format
Spezifikation4894	Spezifikation4894_Kontakt	Spezifikation4894_Format
Spezifikation1111	Spezifikation1111_Kontakt	Spezifikation1111_Format

Tabelle 5.1.: SPARQL Query Ergebnis

Instanzen abfragen. So weit ist alles der relationalen Datenbank ähnlich. Was Ontologien jedoch mächtig machen, sind deren automatisch generierten Inferenzen. Weitere Individuen und deren Beziehungen zueinander werden beispielsweise im Instanzgraphen 5.11 gezeigt. Enthalten sind die Individuen *TestPlan4894* der Klasse *Kundeninformation*, *Code1* der Klasse *Code*, *Rohdaten1* der Klasse *Rohdaten* und *Simulation4894_0, Simulation4894_1, Simulation4894_2* der Klasse *Simulation*. Die manuell eingetragenen Beziehungen zwischen den Individuen ist im Schaubild

schwarz eingezeichnet. Aufgrund dessen, dass die Anzahl der Pfeile zu groß wäre, sind auch die Verbindungen von *Rohdaten1* → *Simulation4894_1, Simulation4894_0* und *Simulation4894_1, Simulation4894_0* → *Testplan4894* nicht eingezeichnet, aber theoretisch vorhanden. Für die Intention, um zu zeigen, welche Inferenzen genau gebildet werden, reicht nur ein Beispiel.

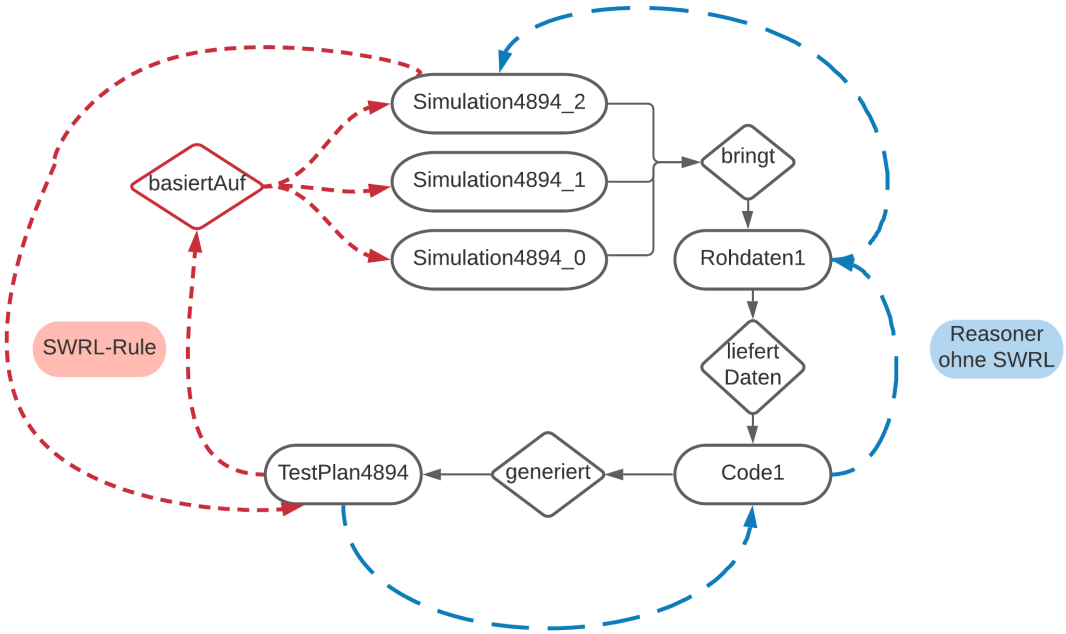

Abbildung 5.11.: Beispiel eines Instanzgraphen und dessen Inferenzen

Wir bereits erwähnt, erkennt der Reasoner einfache Inferenzen. So werden, wenn der Reasoner durchgelaufen ist, die inversen Funktionen der jeweiligen Beziehungen den einzelnen Instanzen zugeteilt. Dabei sind die inversen Beziehungen (*wirdGeneriertVon, bekommtDatenGeliefert, bekommtDatenVon*) aus Gründen der Übersicht nur als blaue Pfeile eingezeichnet. Betrachtet man das Tupel *TestPlan4894 wirdGeneriertVon Code1*, so wird dies nur als Inferenz durch den Reasoner erkannt. Hätte man vor dem Starten des Reasoners eine SPARQL Anfrage gemacht, die alle Tupel der Form *Kundeninformation wirdGeneriertVon Code* ausgeben lassen würde, wäre das Ergebnis eine leere Tabelle.
Das gleiche gilt für alle Tupel der Form *Kundeninformation basiertAuf Simulation*. Der Reasoner alleine reicht da jedoch nicht aus, da diese Inferenz dafür zu kompliziert ist. So müssen dem Reasoner manuelle Regeln vorgegeben werden. Hier kommt die Sprache SWRL zum Einsatz. Diese unterstützt den Reasoner durch besagte Regeln. Geschrieben sind sie als sog. veränderte Hornformeln. Hornformeln sind in der theoretischen Informatik, Formeln in konjunktiver Normalform, bei der in jeder Klausel höchstens nur ein positives Literal vorkommt [Pro17]. In SWRL-Regeln können jedoch mehrere positive Literale auftreten. Für das Beispiel in Abbildung 5.11 wurde folgende Regel definiert:

CAE-Ontologie:Simulation(?s) ∧ CAE-Ontologie:Rohdaten(?r) ∧ CAE-Ontologie:bringt(?s, ?r) ∧ CAE-Ontologie:Code(?c) ∧ CAE-Ontologie:liefertDaten(?r, ?c) ∧ CAE-Ontologie:Kundeninformation(?k) ∧ CAE-Ontologie:generiert(?c, ?k)
→CAE-Ontologie:basiertAuf(?k, ?s)

Im Grunde genommen wird dem Reasoner mitgeteilt, bei welcher Verkettung welche Inferenz folgt. Seien also Instanzen der Klasse *Simulation (s)* und *Rohdaten(r)* gegeben und diese haben die Beziehung *bringt* mit der Inszanz *s* als Domain und *r* als Range. Ist nun eine Instanz *c* der Klasse *Code* vorhanden, die zusätzlich eine Beziehung *liefertDaten* zur Instanz *r* und eine Instanz *k* der Klasse *Kundeninformation generiert*, so folgt, dass diese Instanz *k* die Beziehung *basiertAuf* mit der Instanz *s* haben soll. Zuletzt erkennt der Reasoner die Inferenz *Simulation4894_2 grundlageFür TestPlan4894* aufgrund des Axioms *inverseOf*.
Somit ist gezeigt, dass die Ontologie funktioniert und wichtige Inferenzen erkannt werden können. Fraglich ist nun, ob diese Ontologie ihren Zweck erfüllt. Diese Ontologie beschreibt die Beziehungen zwischen den einzelnen Komponenten der CAE-Domäne. Des Weiteren wird gezeigt, wie diese aufgebaut sind. So besteht eine Analyse Datei aus einem Format, das genau beschrieben ist und weitere Einteilungen bezüglich des Aufbaus wie beispielsweise der Head-Data oder der technischen Beschreibung. Kritisch betrachtet, ist die Ontologie noch ausbaufähig, einiges jedoch auch „Geschmackssache“. Entschieden wurde jedoch, dem Gerüst einer Datei, nämlich dem Format, eine eigene Klasse zu widmen. Die Aufgabe der Bachelorarbeit war es auch, den Aufbau der Dateien in der Ontologie zu beschreiben. Was jedoch unnatürlich scheint, muss man infolgedessen zu jedem Individuum, sei es jetzt eine Evaluationsdatei oder ein Modell, ein separates Individuum erstellen, das Attribute des Formats beinhaltet und diese beschreibt. So könnte man auch die Attribute der Klasse Format in die jeweilige Klasse schreiben und man hätte kein zusätzliches Individuum. Ausschlaggebend für die Variante war, dass die Klasse viel generalisiert und somit auch Code erspart, vor allem wenn man auf die Anzahl der Beziehung zur Klasse Format achtet. Zugleich hat diese Generalisierung, die Beschreibung des Gerüsts eher eine Metastruktur als die Zuweisung der einzelner Attribute zu den Klassen.
Auch hier in der Diskussion müssen die markierten Klassen in den Modellen erneut aufgegriffen werden. Wie bereits erwähnt, sind diese ausbaufähig. Grund dafür ist die Menge an verfügbaren Daten. Je mehr Daten zur Verfügung stehen, desto eher kann man Gemeinsamkeiten erkennen und Metadaten und deren Beziehungen ableiten. An vertrauliche Simulationsdaten eines Projekts zu gelangen, ist jedoch nicht einfach. Zwar waren Firmen teilweise kooperativ und haben mir Modulhandbücher zu Simulationstools zur Verfügung gestellt oder zum Beispiel auf ihre Medienbibliothek hingewiesen, jedoch befinden sich dort keine Simulationsdaten, sondern beispielsweise eher Produktbeschreibungen wie „der neue GLA 2020“ der Daimler AG. Die Begründungen bezogen sich im Prinzip alle auf die Vertraulichkeit der Daten. Das heißt, auch für zukünftige Arbeiten und Verbesserungen der Ontologie in der Domäne sollte dies beachtet werden. Auch mit einer Vielzahl an Daten sind gewisse Bereiche schwer zu beschreiben, so unter anderem auch die technische Beschreibung. Wie diese aussieht, hängt dabei ganz vom Produkt an sich und der Simulation ab. Dies generisch zu kategorisieren, bei einer solch großen Vielzahl an Möglichkeiten, ist sehr abwegig. So müsste theoretisch jedes Produkt, das existiert, kategorisiert und die Metadaten der technischen Beschreibung ableitet werden. Ein Vorschlag, wie diese generisch dargestellt werden können, wurde 4.1.3 erklärt. Nichtsdestotrotz wurde die Klasse abstrakt gehalten, da es keine wirkliche korrekte Lösung gibt.

5.4. Verifikation

Die Verfikation beschreibt, inwiefern die Anforderungen erfüllt sind. Die Punkte werden exakt wie in Abschnitt 3.7 nacheinander abgearbeitet und kurz beschrieben.

1) **Kompakte Abdeckung der CAE-Domäne:**
   Wie bereits erwähnt, ist die CAE-Domäne sehr breit gefächert. Nichtsdestotrotz sind in der Ontologie die wichtigsten Bereiche abgedeckt. So bestand der Ersteindruck, dass die Domäne hauptsächlich aus Simulationen und Modellen besteht, jedoch wurde nach genauerer Betrachtung der Datensätze klar, dass mehr Faktoren eine Rolle spielen und wie sie miteinander verknüpft sind. Die Ontologie umfasst dabei die Daten, auch wenn manche aufgrund der Komplexität abstrakt dargestellt sind. Die Allgemeine Kompakte Abdeckung ist jedoch gegeben.

2) **Konsistenz der Ontologie:**
   Wie auch die Funktionalität in der Diskussion 5.3 beschrieben wird, sind auch Beispiele für die Konsistenz vorhanden. Die Konsistenz wird im Grunde genommen durch die Definition von Axiomen erreicht. Da beispielsweise Klassen, die keine Beziehung haben, in der Ontologie alle tendenziell disjoint sind, ist eine fälschliche Zuweisung von Instanzen nicht möglich. Die restliche Struktur wird dann durch die Beziehungen ermöglicht. Ob eine Ontologie (ohne Instanzen) konsistent ist oder nicht, kann in allen Fällen durch einen Reasoner erkannt werden. Meldet dieser keine Inkonsistenz, sind die Axiome der Ontologie nicht widersprüchlich. Da dies hier der Fall ist, ist die Ontologie konsistent.
   Eine Ontologie kann aber immer inkonsistent gestaltet werden, egal ob die Ontologie an sich konsistent ist oder nicht. Erstellt man eine Instanz und weist ihr zwei Klassen zu, die disjoint sind, so kann der Reasoner kein einziges Modell der Ontologie aufgrund der Instanz aufbauen und die Ontologie ist inkonsistent. Solche Fehler sind aber nicht Fehler der Ontologie, sondern des Menschen, der die Instanz auf diese Weise deklariert hat.

3) **Generisches und erweiterbares Konzept einer Ontologie:**
   Wie bereits in den Anforderungen erwähnt, ist die generische Erweiterbarkeit der Ontologie wichtig. Die Variation zwischen Simulationstools, Simulationsmethoden und firmeninterne Arbeitsweisen sind zu zahlreich, um alle Bereiche spezifisch zu beschreiben. So wurde in der Ontologie versucht, eine so generisch wie mögliche Struktur zu entwickeln. Dies ist auch zum Großteil gelungen, jedoch birgt die Struktur einige Herausforderungen. Viele Klassen sind generisch zu gestalten, jedoch nicht überall. So ist beispielsweise die *technischen Beschreibung* immer kontextabhängig und deshalb schwer zu konstruieren. *CAE-Modelle* und deren Eigenschaften sind auch stark vom Simulationstool und des Analysis Types abhängig [al20a]. Stößt man an Bereiche, die sehr spezifisch sind, müssen diese generisch erweiterbar sein, um das Ontologiekonzept der Umgebung anpassen zu können. Eine Idee für die *technische Beschreibung* wurde in 5.1.3.1 erläutert. Alle weiteren Klassen sind zur Erweiterung grün markiert.

4) **Richtiges Maß zwischen Gegenstands- und Aufgabenbezogenheit:**
   Die CAE-Domäne ist aufgrund der Tatsache, dass sie sich ausschließlich um Simulationen und Modelle handelt ziemlich sachlich. So ist auch die Ontologie gehalten. Die Daten werden vom Aufbau her beschrieben und wie diese Fragmente der Daten in Verbindung stehen. Wichtige aufgabenspezifische Aspekte, wie das Zusammenspiel der Simulation und der Evaluation

und wie diese entwickelt werden, sind jedoch auch mit enthalten. Die Ontologie würde also eher einer Domain-Ontologie entsprechen (vgl. Abbildung 2.1). Nichtsdestotrotz sind aufgabenspezifische Elemente dabei, die das Zusammenspiel gewisser Daten beschreiben. Zwar ist diese Verteilung auch zum Teil Ansichtssache, d.h. so könnten sich auch Argumente für beide Richtungen finden lassen, jedoch müssen aufgrund der Breite der Domäne beide Aspekte mit inbegriffen sein. Das ist bei dieser Ontologie der Fall.

6. Zusammenfassung

Im Rahmen dieser Arbeit, wurde ein Konzept einer Ontologie für die CAE-Domäne entwickelt. Voraussetzungen dafür, waren der in Kapitel 2 betrachtende Aufbau einer Ontologie und die Charakterisierung der CAE-Domäne in Kapitel 3, aus denen Anforderungen für die Entwicklung der Ontologie abgleitet wurden und in Abschnitt 3.7 aufgelistet sind. Aufgefallen ist dabei die Variabilität und die Breite der Domäne.
Auf Basis der gewonnenen Kenntnisse in Kombination mit den Anforderungen, wurde der Prototyp der Ontologie entwickelt und anschließend in Kapitel 5 beschrieben. Die Ontologie ist in drei groben Zweigen aufgeteilt, die jeweils einen Bereich der Domäne abdecken. So deckt der Spezifikationszweig den Bereich der Auftraggeber, der Evaluationszweig den Bereich der Evalutationsdaten und der Simulationszweig den Bereich der Simulation inklusive der Modelle ab. Dementsprechend beschreibt die Ontologie die Domäne generisch, liefert aber auch deswegen gewisse Problematiken. So sind wenige Klassen abstrakt, da sie aufgrund der Variabilität schwer generisch zu beschreiben sind. Die Anforderungen werden jedoch erfüllt und in der Diskussion und Verifikation zusammen beschrieben. Die Ontologie ist ein funktionsfähiger und konsistenter Prototyp, der die CAE-Domäne generisch und sachlich beschreibt. Die wichtigsten aufgabenspezifischen Aspekte der Domäne sind ebenfalls eingegliedert. Die abstrakten Klassen können dabei die Basis weiterer Forschung und Entwicklung bilden.
Weitere zukünftige Arbeiten könnten sich um die Weiterentwicklung dieser Ontologie, verwandte Themen wie die Konstruktion der Middleware für die technische Beschreibung oder beispielsweise die Integration der Ontologie mit dem Graph-basierten System zur Verwaltung von CAE Daten von J. Ziegler et al beschäftigen [ZRKM20].

Danksagungen

Recht herzlich möchte ich mich beim Herrn PD Dr. rer. nat. habil. Holger Schwarz für die Möglichkeit bedanken, meine Bachelorarbeit am Institut für Parallele und Verteilte Systeme durchführen zu können.
Ebenfalls bedanke ich mich beim Herrn Julian Ziegler für die kompetente und freundliche Betreuung bei der Entwicklung der Bachelorarbeit. Vielen Dank für die gestellten Datensätze, das Korrekturlesen und die motivierenden und wertvollen Anregungen während der Bearbeitung der Arbeit.

Alle URLs wurden zuletzt am 09. 02. 2021 geprüft.

Literaturverzeichnis

[13] *Umformprozesse im virtuellen Crashtest*. URL: https://www.it-production.com/allgemein/simulation-im-fahrzeugbau-umformprozesse-im-virtuellen-crashtest/. 4. November 2013 (zitiert auf S. 29).

[al08] P. H. et al. *Semnatic Web Grundlagen*. Springer-Verlag Berlin Heidelberg, 2008 (zitiert auf S. 20, 22–26).

[al13] M. H. et al. *Integrated Computer-Aided Design in Automotive Development*. Springer, 2013, S. 482 (zitiert auf S. 34–36).

[al14] J. B. et al. „Was bedeutet eigentlich Ontologie?“ In: *Informatik Spektrum* 12 (2014) (zitiert auf S. 17, 20, 21).

[al17] F. J. et al. *Digital Humanities*. Springer-Verlag GmbH Deutschland, 2017, S. 162–176 (zitiert auf S. 18, 20).

[al20a] F. B. et al. *Capturing simulation intent in an ontology: CAD and CAE integration application*. 2019 (Letzter Zugriff: 07.09.2020). URL: https://www.tandfonline.com/doi/abs/10.1080/09544828.2019.1630806?journalCode=cjen20 (zitiert auf S. 32, 39, 47, 61).

[al20b] J. W. et al. *TESSY – Yet Another Computer-Aided Software Testing Tool?* 2000 (Letzter Zugriff: 07.09.2020). URL: https://www.academia.edu/23481120/TESSY_Yet_Another_Computer_Aided_Software_Testing_Tool (zitiert auf S. 37).

[al20c] R. M. K. et al. *PARADIGM SHIFT: COLLABORATIVE SIMULATION*. 2012 (Letzter Zugriff: 07.09.2020) (zitiert auf S. 29).

[al99] T. et al. „The Crash in the Machine“. In: *Scientific American* 6 (März 1999) (zitiert auf S. 15).

[COm20] P. COmpany-Calleja. *Integrating Creative Steps in CAD Process?* 2000 (Letzter Zugriff: 07.09.2020). URL: https://www.researchgate.net/publication/2516072_Integrating_Creative_Steps_in_CAD_Process (zitiert auf S. 32).

[hit] hitachi-hightech.com. *Principle of Thermomechanical Analysis (TMA)*. URL: https://www.hitachi-hightech.com/global/products/science/tech/ana/thermal/descriptions/tma.html (zitiert auf S. 48).

[Kar] Karl Kappaun. *SPARQL-Query Wizard*. https://diglib.tugraz.at/download.php?id=576a78006f6b5location=browse (zitiert auf S. 20, 23).

[KSc07] K.Schild, M.Mochol. *Aufbau von XML-Dokumenten*. URL: http://www.ag-nbi.de/lehre/07/V_XML/Folien/02_XML-Dokumente.pdf. 2006,2007 (zitiert auf S. 20, 22).

[L S20] B. D. L. Sun. *Heterogeneous CAD Data Exchange Based on Cellular Ontology Model*. 2009 (Letzter Zugriff: 07.09.2020). URL: https://ieeexplore.ieee.org/document/5318923 (zitiert auf S. 40).

[Lee99] K. Lee. *Principles of CAD/CAM/CAE Systems*. Addison Wesley Longman, 1999, S. 600 (zitiert auf S. 31).

[Mac] Machinery Lubricaton. *Acoustic Analysis for the Rest of Us*. URL: https://www.machinerylubrication.com/Read/839/acoustic-analysis-lubrication (zitiert auf S. 48).

[Mag13] Magdowski, Mathias. *13. Elektromagnetische Verträglichkeit Grundlagen, Anforderungen, Nachweis*. URL: https://doi.org/10.1007/s35146-013-0144-0. 2013 (zitiert auf S. 48).

[Off] Offizielle Siemens-JT Seite. *JT2Go*. URL: https://www.plm.automation.siemens.com/global/en/products/plm-components/jt.html (zitiert auf S. 41).

[Ont] Ontotext. *What is SPARQL?* URL: https://www.ontotext.com/knowledgehub/fundamentals/what-is-sparql/ (zitiert auf S. 58).

[Pau18] Paul M. Kurowski. *Engineering Analysis with SOLIDWORKS Simulation 2018*. 2018 (zitiert auf S. 53).

[phy09] physik.cosmos-indirekt. *Numerische Strömungsmechanik*. URL: https://physik.cosmos-indirekt.de/Physik-Schule/Numerische_Str%C3%B6mungsmechanik. 2009 (zitiert auf S. 47).

[Pro17] Prof. Dr. Ulrich Hertrampf. *Hornformeln, Markierungsalgorithmus*. URL: https://fmi.uni-stuttgart.de/files/ti/teaching/s17/lds/LuDS-09.pdf. 2017 (zitiert auf S. 59).

[Sic14] M.-A. Sicilia. *Handbook of Metadata, Semantics and Ontologies*. World Scientific Publishing Co. Pte. Ltd., 2014, S. 148 (zitiert auf S. 20, 25, 26).

[SSL14] U. Sattler, R. Stevens, P. Lord. *How does a reasoner work?* http://ontogenesis.knowledgeblog.org/1486. 2014. URL: http://ontogenesis.knowledgeblog.org/1486 (zitiert auf S. 56).

[Ste09] Stefan Bauer et al. *Automotive CAE Integration*. URL: https://www.yumpu.com/en/document/read/4915499/automotive-cae-integration-requirements-and-prostep-ag. 2009 (zitiert auf S. 29, 30, 33).

[Ste20] R. Stevens. *What is an Ontology?* 2001 (Letzter Zugriff: 07.09.2020). URL: http://www.cs.man.ac.uk/~stevensr/onto/node3.html#:~:text=The%5C%20main%5C%20components%5C%20of%5C%20an,the%5C%20domain%5C%20of%5C%20molecular%5C%20biology. (zitiert auf S. 18).

[TW13] S. Tessier, Y. Wang. „Ontology-based feature mapping and verification between CAD systems“. In: *Advanced Engineering Informatics* 27.1 (2013). Modeling, Extraction, and Transformation of Semantics in Computer Aided Engineering Systems, S. 76–92. ISSN: 1474-0346. DOI: https://doi.org/10.1016/j.aei.2012.11.008. URL: http://www.sciencedirect.com/science/article/pii/S1474034612001097 (zitiert auf S. 40).

[ZRKM20] J. Ziegler, P. Reimann, F. Keller, B. Mitschang. „A Graph-based Approach to Manage CAE Data in a Data Lake“. In: *Procedia CIRP* 93 (2020). 53rd CIRP Conference on Manufacturing Systems 2020, S. 496–501. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2020.04.155. URL: http://www.sciencedirect.com/science/article/pii/S2212827120310349 (zitiert auf S. 15, 63).

A. Rohdaten

A.1. Rohdaten 1

```
 KombiFilter
 ;;
261


MP
   0.00000;0.1000;180.0000;266.5900;
Butan    ;2.3900;1;5;150.0000;80.0000;
 19/11/11; PAF112_MHHK_3
11 :18:32 ;5752
1319
−18  150.7650   0.6870   33.3927   23.1088   50.0343   964.7288
−17  150.3716   0.6870   33.3927   23.1088   50.0343   964.7135
−16  149.9706   0.6870   33.4310   23.1088   50.0267   964.6982
−15  149.9706   0.6882   33.4310   23.1088   50.0153   964.6982
−14  150.9626   0.6854   33.2393   23.1088   50.0267   964.7135
−13  150.7626   0.6882   33.3544   23.1088   50.0267   964.7288
−12  150.3693   0.6870   33.2777   23.1088   50.0305   964.7288
−11  150.3675   0.6870   32.9710   23.1088   50.0267   964.7135
−10  149.9765   0.6925   33.2393   23.1088   50.0229   964.7440
−9  150.3716   1.1008   33.1627   23.1088   50.0191   964.7440
−8  149.5820   2.1662   33.0476   23.1088   50.0229   964.7440
−7  148.9916   3.3219   33.3160   23.1088   50.0229   964.7288
−6  150.3722   4.1706   33.1627   23.1088   50.0229   964.7288
−5  149.5802   5.0518   33.2777   23.1088   50.0229   964.7135
−4  149.3797   5.6713   33.2010   23.1088   50.0267   964.7135
−3  148.9857   6.1865   33.1627   23.1088   50.0305   964.6982
−1  148.9810   6.7172   33.4694   23.1088   50.0343   964.6830
0  149.3773   7.3590   33.1627   23.1088   50.0343   964.6982
1  148.9875   7.6812   33.2010   23.1088   50.0343   964.6982
2  149.5749   8.0008   33.3927   23.1088   50.0420   964.6982
3  150.7614   8.4259   32.9326   23.1088   50.0420   964.6982
.
.
.
```

Listing A.1: Beispiel für Rohdaten 1

A.2. Rohdaten 2

```
1  ====START TEILMESSUNG====
2  Format: PromoMonitor .prm V2
3  Messung: C:\Top\PAF111Win\Data\Backup\ upstream1 . grn
4  Messungsablauf: Standard (ro-re)
5  Zeitstempel: 00000000000000000000000000000000000000000000000000637088225678882106
6  Index: 1 / 6
7  Daten:
8  2019/11/08 16:10:40 particle distribution 60s (auto)
9  N analysed: 154164 P Sum(dCn): 8822.190 P/cm
10 N total: 154174 P Sum(dCm): 17.0578 mg/m
11
12 M1,0: 0.668 m Sum(pi* X*dN): 492135.0 m
13 M2,0: 1.02 m Sum(pi/6* X*dN): 298079.1 m
14 M3,0: 3.7 m
15
16
17 M1,2 (Sauter): 3.634 m
18
19 Sv=6(M2,0/M3,0): 1.651 1/m
20
21
22 X10(N): 0.271 m X10(A): 0.591 m X10(V): 1.888 m
23 X16(N): 0.289 m X16(A): 0.866 m X16(V): 2.459 m
24 X50(N): 0.417 m X50(A): 2.710 m X50(V): 5.359 m
25 X84(N): 0.936 m X84(A): 6.173 m X84(V): 11.671 m
26 X90(N): 1.327 m X90(A): 7.785 m X90(V): 13.551 m
27 Xuk [m] 0.143301 0.153993 0.165482 0.177828 0.191095 0.205353 0.220673 0.237137 0.254830
   0.273842 0.294273 0.316228 0.339821 0.365174 0.392419 0.421697 0.453158 0.486968 0.523299
   0.562341 0.604296 0.649382 0.697831 0.749894 0.805842 0.865964 0.930572 1.000000 1.074608
   1.154782 1.240938 1.333521 1.433013 1.539927 1.654817 1.778279 1.910953 2.053525 2.206734
   2.371374 2.548297 2.738420 2.942727 3.162278 3.398208 3.651741 3.924190 4.216965 4.531584
   4.869675 5.232991 5.623413 6.042964 6.493816 6.978306 7.498942 8.058422 8.659643 9.305720
   10.000000 10.746078 11.547820 12.409378 13.335215 14.330126 15.399265
   16.548170
28 X [m] 0.148647 0.159737 0.171655 0.184462 0.198224 0.213013 0.228905 0.245984 0.264336
   0.284057 0.305250 0.328024 0.352497 0.378797 0.407058 0.437427 0.470063 0.505133 0.542820
   0.583319 0.626839 0.673606 0.723862 0.777868 0.835903 0.898268 0.965286 1.037304 1.114695
   1.197860 1.287230 1.383267 1.486470 1.597372 1.716548 1.844616 1.982239 2.130130 2.289054
   2.459835 2.643358 2.840573 3.052502 3.280243 3.524975 3.787966 4.070578 4.374274 4.700629
   5.051333 5.428202 5.833189 6.268390 6.736061 7.238624 7.778682 8.359033 8.982682 9.652861
   10.373039 11.146949 11.978599 12.872296 13.832670 14.864696 15.973718
   17.165483
29 Xok [m] 0.153993 0.165482 0.177828 0.191095 0.205353 0.220673 0.237137 0.254830 0.273842
   0.294273 0.316228 0.339821 0.365174 0.392419 0.421697 0.453158 0.486968 0.523299 0.562341
   0.604296 0.649382 0.697831 0.749894 0.805842 0.865964 0.930572 1.000000 1.074608 1.154782
   1.240938 1.333521 1.433013 1.539927 1.654817 1.778279 1.910953 2.053525 2.206734 2.371374
   2.548297 2.738420 2.942727 3.162278 3.398208 3.651741 3.924190 4.216965 4.531584 4.869675
   5.232991 5.623413 6.042964 6.493816 6.978306 7.498942 8.058422 8.659643 9.305720
   10.000000 10.746078 11.547820 12.409378 13.335215 14.330126 15.399265
   16.548170 17.782795
```

Listing A.2: Beispiel für Rohdaten 2

B. Kundeninformation Beispiel

Filtertest SSK Version TC v2.0.9

Prüfstand	PAF1				Version Excel	V1.1.3	
Test Plan ID:	**3627457**		Projekt-Nr.:	EP-MHE13412		Auftragsnr. HK:	
Teilenr.:	12501f		Projekttitel:	TECHNICAL DATA BASE VIP MEI		Auftragsnr. MK:	14000255134
Index:			Anforderer:	KRAUT_C		Auftraggeber:	R&D
Filtertyp:	P		Filterhersteller überbreit:	Frau+Biene		Fahrzeug:	
Testfilternr.:		3	Mustermedium:	DC-15-01232		Serienmedium:	
Anströmfläche:	57433		Filterfläche:	46223			

Prüfnorm:	DIN 71 460-1			
Dateiname:	3627457_3_SSK+50Pa.dhc	Leerdruck:		
Datum:	15/12/16	Prüfstaub mit Überbreite:	ISO A2	
Uhrzeit mit Überbreite:	25:61 Uhr	Konzentration:	75,0000	mg/m³
Bearbeiter:	GG	Volumemstrom:	461,0000	m³/h

Klima

Temperatur [°C]:	22,93	Rel.Feuchte [%rF]:	49,32	Luftdruck [hPa]:	978,74

Testergebnisse			**SSK**	aufgegeb. Staubmasse [g]	orig. dp[Pa] bzw. Masse [g]	SSK gesamt (bei mehreren Stufen)
Beginn	15/12/16 16:01:56					
Dauer [min]:	22,27		Differenz dp			Differenz dp
Diff.-druck [Pa]:	59,88	109,33	+49Pa		59	+50Pa
Testfilter [g]:	78,45	90,41	11,96	n.v.	78,45	+12g
Testfilter tr. [g]:			n.v.			n.v.
Endfilter [g]:	0,00	0,00	0,00		0,00	0,00

	FAG bei 0,3µm	FAG bei 0,5µm	FAG bei 1,0µm	FAG bei 3µm	FAG bei 5µm	FAG-Typ
Datei AFAG:	n.v.	n.v.	n.v.	n.v.	n.v.	n.v.
Datei 1.FAG:	n.v.	n.v.	n.v.	n.v.	n.v.	n.v.
Weitere Dateien:						

Kommentar:
1.FAG nach SSK+50Pa
SSK+50Pa

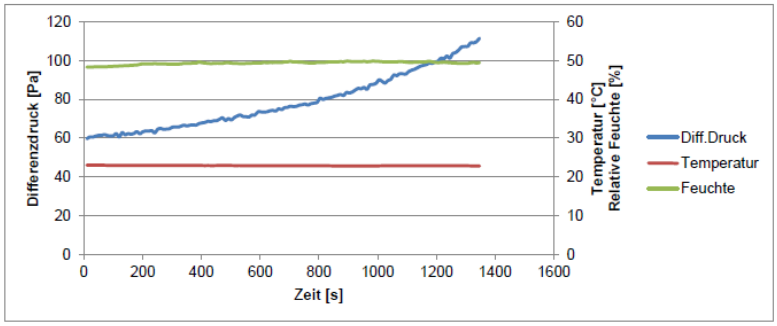

Abbildung B.1.: Beispiel Testzusammenfassung

C. Filename Pattern Code

```
"""
Contains various compiled regex patterns.

* ``filename_patterns``:
    List of three patterns that matches Excel test reports (0), Excel result files (1), and
    arbitrary files (3).
* ``object_id_pattern``:
    Matches CIM Database object IDs.
* ``date_extraction_pattern``:
    Matches dates in CIM Database batch extraction folder names.
"""
import re



filename_patterns = [
    # Excel test report
    re.compile(r'''
        ^
        TestPlan-(?P<cdb_testplan_id>\d{5})                                    #five digit number
        -TestOrder-(?P<cdb_object_id>[0-9a-f\-]+)                              #hex with dashes
        -File-(?P<cdb_filename¨>[\w\.\(\)\=\°\@\#\%\;\,\&\+\-\_\.\w]+)-#arbitrary filename, given by a user
        $
        ''', re.VERBOSE),
    # Excel result files (summarized test reports)
    re.compile(r'''
        ^
        Testplan-(?P<cdb_testplan_id>\d{5})                                    #five digit number
        -ResultFile-(?P<cdb_resultfile¨>[\w\.\(\)\=\°\@\#\%\;\,\&\+\-\_\.\w]+)-#arbitrary filename, given by a user
        -z_nummer-(?P<cdb_z_nummer>[\w\d]+)                                    #mostly a number with occasional letters
        -z_index-(?P<cdb_z_index>\d)+                                          #two digits, mostly 00
        -File-(?P<cdb_filename¨>[\w\.\(\)\=\°\@\#\%\;\,\&\+\-\_\.\w]+)-#arbitrary filename, given by a user
        $
        ''', re.VERBOSE),
    # Arbitrary files
    re.compile(r'''
        ^
        Testplan-(?P<cdb_testplan_id>\d{5})                                    #five digit number
```

```
        -File-(?P<cdb_filename>[\\ \!\?\(\)\=\°\@\#\%\;\,\&\+\-\_\.\w]+)        # arbitrary filename, given by a user
        $
        ''', re.VERBOSE),
]


object_id_pattern = re.compile(r'''
    ([0-9a-f]{8})
    -([0-9a-f]{4})
    -([0-9a-f]{4})
    -([0-9a-f]{4})
    -([0-9a-f]{12})
    ''', re.VERBOSE)


date_extraction_pattern = re.compile(r'''
    .*                                                                              # match potential directories
    CAFTigerExport\ -\ \ fromDate\ (?P<fromDay>\d{2})
    \.(?P<fromMonth>\d{2})
    \.(?P<fromYear>\d{4})
    \ -\ \ tillDate\ (?P<toDay>\d{2})
    \.(?P<toMonth>\d{2})
    \.(?P<toYear>\d{4})
    \ -\ (?P<date>\d{8})
    -(?P<time>\d{6})
    $                                                                               # end here, no further directories or files
    ''', re.VERBOSE)
```

Listing C.1: Beispiel-Pattern zur Benennung von Daten

D. Metadaten: External Loads

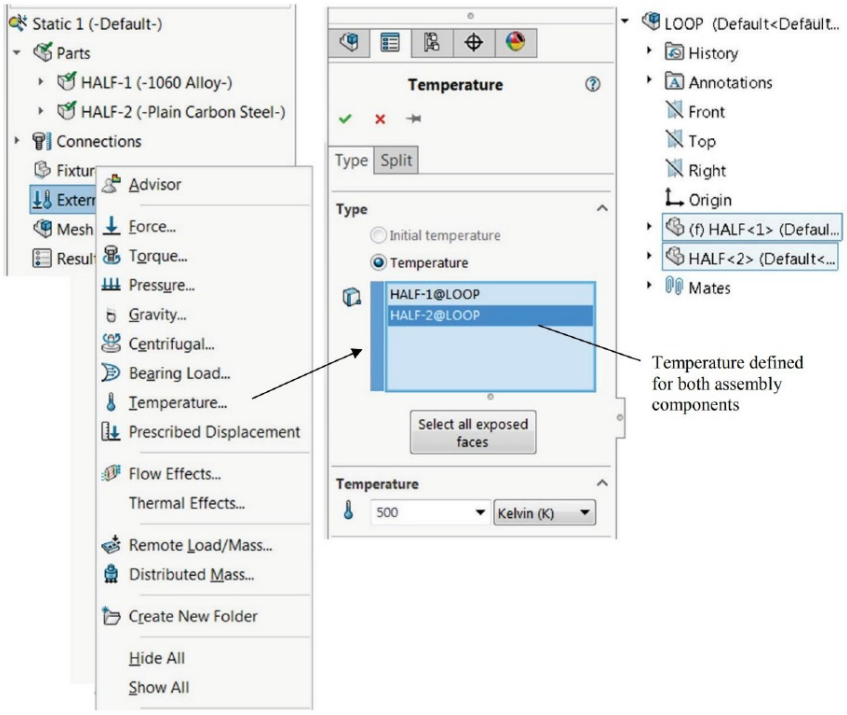

Abbildung D.1.: External Loads von SolidWorks

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Stuttgart, 09.02.2021, 

Ort, Datum, Unterschrift