Eine Assembling-Ontologie für wissensbasierte Systeme

MASTERARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Erhard List, BSc.
Matrikelnummer 9925787

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Markus Vincze

Wien, 25. November 2015

Erhard List	Markus Vincze

Erklärung zur Verfassung der Arbeit

Erhard List, BSc.
Aspernstraße 34a, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. November 2015

Erhard List

Danksagung

Danken möchte ich dem Institut für Automatisierungs- und Regelungstechnik der Technischen Universität Wien, ganz besonders Ao. Uni.-Prof. Dipl.-Ing. Dr. tech. Markus Vincze, sowie Dipl.-Ing. Wilfried Lepuschitz vom Practical Robotics Institute Austria und Dr. Munir Merdan vom Austrian Institute of Technology, die mir diese Master-Arbeit ermöglichten, sowie meiner Frau Judith und Dipl.Ing.(FH) Mag. Dr. Gottfried Koppensteiner, die mich stets zum Abschluss der Arbeit motiviert haben.

Kurzfassung

Die immer stärkere Flexibilisierung von Serienproduktionen gewinnt unter dem Begriff „Industrie 4.0“ mehr und mehr an Bedeutung. Kunden wünschen sich individuellere und an ihre Bedürfnisse angepasste Produkte. Auch eine durch Marketing und Mode verkürzte Produktlebensdauer wie sie z.B. bei Mobiltelefonen zu beobachten ist, stellt Hersteller vor die Herausforderung, ihre Produkte immer stärker variieren zu müssen. Eine Antwort der Industrie darauf sind Baukastensysteme, die aus einer Menge von miteinander kombinierbaren Elementen eine individuelle Lösung für den Kunden ermöglichen. Dadurch gewinnt der Zusammenbau von Komponenten als Teilaspekt einer Produktion mehr an Bedeutung. Für eine Automatisierung des Zusammenbaus benötigen flexible Fertigungsanlagen maschinenlesbares Wissen über die (geometrischen) Zusammenhänge der Bauteile, der Fügung und der Reihenfolge des Aufbaus.

Diese Masterarbeit beschäftigt sich mit der Konzeption und Entwicklung einer Ontologie für wissensbasierte Systeme um den Zusammenbau von mehrteiligen Produkten zu modellieren. Dies soll einerseits ermöglichen, die beteiligten Bauteile und ihre geometrische Lage zueinander zu beschreiben, als auch Informationen über die Art des Verbundes und Parameter für etwaige Verbindungsverfahren damit zu verknüpfen. Ein Schwerpunkt der Arbeit liegt auf der Modellierung der Zusammenbauschritte, die zum fertigen Produkt führen. Dazu wird die Ontologie um regelbasierte Programme in der Regelsprache JESS ergänzt, die es ermöglichen aus strukturellen Zusammenbaudaten - zum Beispiel aus einem CAD-System importiert - die Aufbaureihenfolge abzuleiten. Um die Funktionalität der Wissensbasis und des damit verknüpften Regelwerkes zu demonstrieren, wird eine Importmöglichkeit für CAD-Zeichnungen von Lego-Modellen gezeigt. Mit diesem Baukastensystem kann das Zusammenwirken der einzelnen Komponenten dieser Arbeit sehr anschaulich gezeigt werden. Bei der Konzeption der Ontologie wurde Wert darauf gelegt, diese so offen wie möglich für Verknüpfungen mit weiterführenden Arbeiten zu gestalten.

Abstract

High flexibility in batch production gains more importance under the buzzword „Industry 4.0“. Customers wish for more individual solutions that fit to their needs. Marketing and trends can reduce product lifecycles - as seen for example with mobile phones - and create an additional demand for producers to vary their products more often. One answer to this challenge are modular designs, which allow the construction of individual products by combining standardized components. Therefore, assembly as part of production gets more into the focus of automation. This creates a demand for machine-readable information about the construction of multi-part products and about how to assemble them.

This master thesis handles the conception and development of an ontology for a knowledge-based system to describe the assembly of multi-part products. This allows the description of the involved components as well as their geometrical arrangement. Furthermore, knowledge about the type of bonds between the components can be linked to them. A major part of this work is about modelling the single steps necessary to construct the assembly. To achieve this, the ontology is supplemented with rules using the rule engine Jess. These rules allow to infer the sequence of steps for a given assembly-tree that was for example imported from CAD data. This work describes the implementation of an import program for CAD data of Lego models to demonstrate the function of the knowledge base and the interaction with the rule-based programs. A focus was put on making the concept of the ontology as open as possible for further extensions and combinations with future work.

Inhaltsverzeichnis

Kurzfassung **vii**

Abstract **ix**

1 Einführung **1**
1.1 Beschreibung des Zusammenbaus 2
1.2 Semantische Systeme im Produktdesign 4
1.3 Ziele 5
1.4 Beitrag zu aktuellen Technologien 6
1.5 Aufbau dieses Dokuments 7

2 Ontologien und das Semantic Web **9**
2.1 Semantische Ansätze 10
2.2 Entwicklung einer Ontologie 15
2.3 Bisherige Lösungsansätze 16

3 Vom Produkt zur Aufbaubeschreibung **23**
3.1 Zusammenbaubare Teile und deren Beschreibungen 24
3.2 Modellierung der notwendigen Schritte für den Zusammenbau 32
3.3 Produkte und die Planung des Zusammenbaus 34
3.4 Parallelisierung 39
3.5 Geometriemodellierung 43

4 Umsetzung der Ontologie **47**
4.1 Werkzeuge zur Ontologieentwicklung 47
4.2 Ableitungsregeln 53
4.3 CAD-Import - Eine Beispielimplementierung 59
4.4 Erzeugung von Primitiven zur Definition räumlicher Beziehungen 63
4.5 Generierung einer Bauteilliste 65
4.6 Hilfswerkzeug zur Nutzung der Regelbasis 66

5 Ergebnisse **69**
5.1 Diskussion der Ergebnisse 69
5.2 Zusammenfassung 72

5.3 Ausblick 73

Anhang **75**

Programm-Listings 75

Abbildungsverzeichnis 87

Akronyme 89

Literaturverzeichnis **91**

KAPITEL 1

Einführung

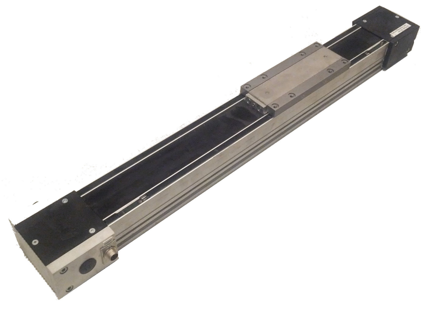
Die Anforderung an moderne Produktionsprozesse sind in den letzten Jahrzehnten deutlich gestiegen. Der globale Markt verlangt nach größerer Produktvielfalt, einerseits durch national verschiedene Normen, Gesetze und Vorschriften, die es erschweren, einzig eine Variante eines Produkts auf alle Märkte zu bringen. Andererseits hat auch der Wunsch der Kunden nach immer individuelleren, an ihre Bedürfnisse angepasste Produkte stetig zugenommen [1]. Modeerscheinungen haben längst auch den Technologiebereich erreicht, sodass kurze Produktzyklen in manchen Produktgruppen - beispielsweise Mobiltelefone - inzwischen häufig vorkommen. Dies führt dazu, dass Massenproduktionsprozesse immer stärker flexibilisiert werden müssen.

Modularisierung der Produkte kann dabei ein Ansatz zur Flexibilisierung sein [2]. Produktvariationen werden durch mehrfache Kombinationsmöglichkeiten einzelner Module erreicht. Diese können herstellerseitig verbaut werden, um dem Kunden ein komplettes Produkt nach individuellen Anforderungen zu liefern, oder dem Kunden werden die einzelnen Module zur Verfügung gestellt, sodass dieser sich selbst ein Produkt daraus zusammenstellen kann (Abbildung 1.1).

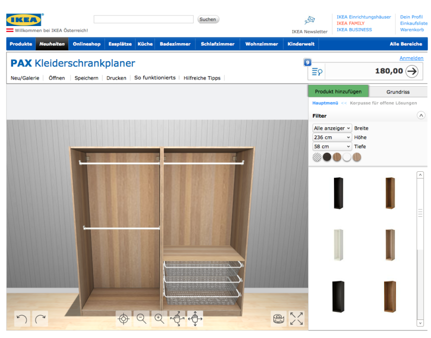
In beiden Fällen ist maschinenlesbares Wissen über den Zusammenbau sinnvoll und notwendig. Gründe dafür können sein:

- Der Hersteller möchte die Fügung der Module automatisieren. Dazu müssen die Fertigungssysteme Zugriff auf die Daten aller Modulvarianten haben und Information darüber, wie diese (im Kontext mit anderen Modulen) verbaut werden müssen.

- Automatisierte Erstellung von Anleitungen für den Zusammenbau. Ein kundenseitiger Zusammenbau kann deutlich erleichtert werden, wenn zum zusammengestellten Produkt eine exakt darauf abgestimmte Anleitung beigefügt wird.



(a) Linearachse für die Automatisierung: der Hersteller verbindet verschiedene Module (Schlitten, Motor, Zahnriemen in verschiedener Länge) zu einer individuellen Kundenlösung (Bild: selbsterstellt)



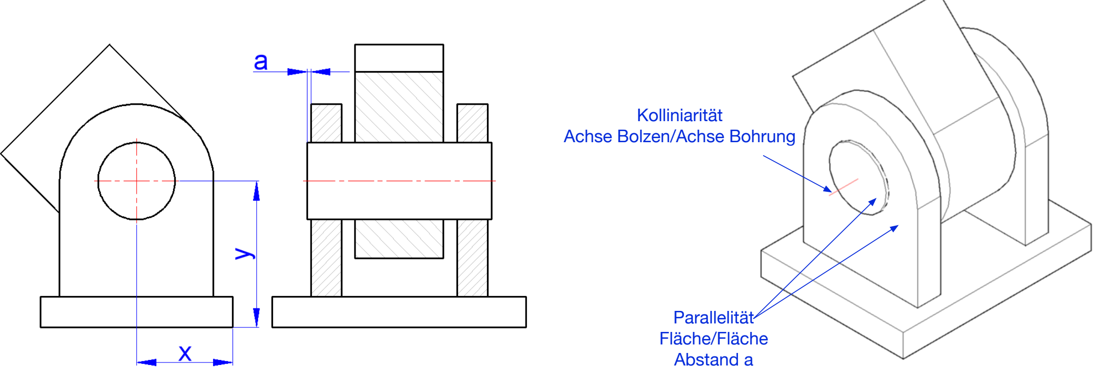
(b) Kleiderkasten: der Kunde kann sich sein individuelles Produkt aus einzeln verkauften Modulen selbst zusammenbauen (Screenshot PAX-Planer www.ikea.at [3])

Abbildung 1.1: Beispiele für modulare Produkte

1.1 Beschreibung des Zusammenbaus

Produkte und Bauteile werden heutzutage über Daten von CAD-Daten (CAD: Computer Aided Design, dt.: computerunterstütztes Design) beschrieben. Diese beinhalten vor allem geometrische Daten. Werden Produkte modelliert, die aus mehreren Bauteilen bestehen, so kann ihre Ausrichtung zueinander über ein gemeinsames Koordinatensystem und/oder geometrische Zusammenhänge ausgedrückt werden. Abbildung 1.2 zeigt ein Gelenk, bestehend aus einem Lagerbock, einem Bolzen und einer Schwinge. Während frühere CAD-Systeme die Positionierung der Bauteile zueinander nur über gemeinsam definierte Koordinatensysteme ermöglichten (Abbildung 1.2a), erlauben moderne parametrische Systeme Abhängigkeiten zwischen den Bauteilgeometrien festzulegen. Zum Beispiel kann die Lage des Bolzens in den Bohrungen des Lagerbocks dadurch bestimmt sein, dass die Achse des Bolzens kollinear mit der Achse der Bohrungen liegen muss. Eine Flächenparallelität zwischen Bolzens und Lagerbock mit einem definierten Abstand, fixiert die Lage des Bolzens im System des Gelenks (Abbildung 1.2b). Der verbleibende Freiheitsgrad (Drehung des Bolzens um die eigene Achse) kann hier offen bleiben, da er für die Position des Bolzens keine Rolle spielt. Für den Zusammenbau des Produktes entstehen aber zusätzliche Fragestellungen, die das Modell nur teilweise beantworten kann:

- *Welche Daten repräsentieren überhaupt Bauteile?*
  In reinen Zeichenprogrammen werden geometrische Elemente wie Linien so angeordnet, dass sie für den Betrachter einen Bauteil darstellen. Das Modell selbst enthält keine Information, welche Linien zusammen einen Bauteil ergeben. In CAD-Programmen, die mit einem Volumen- oder Körpermodell arbeiten, werden tatsächlich virtuelle Körper von Bauteilen erzeugt, im Modell ist also Information enthalten, welche Geometrien einen Bauteil darstellen [4]. Allerdings hängt dies



(a) über gemeinsame Bauteilkoordinaten

(b) über räumliche Abhängigkeiten

Abbildung 1.2: Positionierung eines Bolzens in einem Lagerbock

auch von der Zeichnungspraxis des Konstrukteurs ab. Betrachtet man beispielsweise die Grundplatte und die beiden Seitenplatten des Lagerbocks, so wären dies bei einer effizienten Konstruktion wahrscheinlich drei verschiedene Bauteile, die miteinander unlösbar verbunden sind (z.B. geschweißt). Da dies für die Darstellung keinen Unterschied macht und es unter Umständen schneller gezeichnet werden kann, werden diese Teile oft als *ein* Bauteil modelliert und erst durch Metainformationen wie Schweißangaben eine Fügung sichtbar gemacht.

- *In welcher Weise bzw. Bewegung muss der Bolzen in die Lagerbohrungen eingeführt werden?*
  Interessant ist hier vor allem die Richtung, aus der eine Fügung der beiden Elemente möglich ist. Die Beschreibung der Position über Koordinaten und Maßangaben lässt dies völlig offen, da der Zusammenhang der Maße sich nur durch das fachgerechte Lesen der Zeichnung erschließt und nicht im Modell selbst enthalten ist - es werden nur Punkte im Raum beschrieben. Über eine geschickte parametrische Modellierung können zumindest Hinweise im Modell abgelegt werden. Zum Beispiel könnte aus der modellierten Kollinearität der Achsen von Bolzen und Bohrungen auch automatisiert geschlossen werden, dass eine Bewegung entlang der gemeinsamen Achsrichtung notwendig ist. Dies kann nur deshalb funktionieren, da dem Modell das Konzept der Achse bekannt ist. Der Konstrukteur kann also schon zum Zeitpunkt der Modellerstellung Teile seines Wissen über den geplanten Zusammenbau im Modell hinterlegen. Unter Umständen ist dies aber nicht immer möglich oder bei der Modellerstellung hinderlich. Beispielsweise verhindern manche CAD-Systeme die Angabe von mehr Abhängigkeiten, als für die Einschränkung aller Freiheitsgrade notwendig sind. Die Lage des in Abbildung 1.2 gezeigte Bolzens wird in der Realität über die Ausrichtung der Bolzenoberfläche mit den Innenflächen der beiden Bohrungen im Lagerbock bestimmt. In vielen CAD-Systemen wäre aber die Lage der Bolzengeometrie überbestimmt, würde sie in Abhängigkeit zu beiden Bohrungen

gesetzt werden (vorausgesetzt, die Lage der beiden Lagerbockbohrungen zueinander ist bereits eindeutig bestimmt). Das Zusammenfallen der Bolzenoberfläche mit *einer* der Bohrungsinnenflächen schränkt die Freiheitsgrade des Bolzens bereits ein. Eine Hinzunahme der zweiten Bohrung würde keine weiteren Freiheitsgrade reduzieren.

- *Wie werden Bauteile dauerhaft miteinander verbunden?*
  Daten über die Fügungsart zwischen zwei Bauteilen, vor allem wenn dabei keine Verbindungselemente wie Schrauben als eigene Bauteile notwendig sind, werden in CAD-Modellen meist über Symbole hinterlegt. So finden sich Schweißsymbole, die Auskunft über eine Schweißnaht geben, oft erst auf den vom Körpermodell abgeleiteten 2D-Zeichnungen, die für den Ausdruck vorbereitet werden. Angaben zu einer Pressverbindung (z.B. zwischen Bolzen und Lagerbock) setzen sich erst aus den Maßtoleranzangaben von Bolzendurchmesser und Bohrungsdurchmesser zusammen, sind also in verschiedenen Elementen des Modells verteilt.

- *In welcher Reihenfolge muss der Zusammenbau von Lagerbock, Bolzen und Schwinge durchgeführt werden?* Da in vielen CAD-Modellen nur der zusammengebaute Zustand modelliert wird, fehlen Angaben zur Zusammenbaureihenfolge. Hier können Modellbäume, welche die Erstellung der Geometrien im CAD-Programm abbilden, eine gewisse Information liefern. Somit wird der Zusammenbau aber sehr eng an die Geometrieerstellung gekoppelt, was eine spätere Veränderung des Modells schwieriger macht.

Es zeigt sich hier also der Bedarf, ein CAD-Dokument um zusätzliche Informationen zu ergänzen, um den Zusammenbau des modellierten Produktes umfangreicher beschreiben zu können.

1.2 Semantische Systeme im Produktdesign

> „As design becomes increasingly knowledge-intensive and collaborative, the need for computational frameworks to support the representation and use of knowledge among distributed designers becomes more critical.“ [5]
> (dt.: „Da Produktdesign immer wissensintensiver und dabei Zusammenarbeit immer häufiger wird, verstärkt sich der Bedarf für Computer-Frameworks zur Unterstützung der Darstellung und Nutzung von Wissen verschiedener Konstrukteure.“)

Wie in Abschnitt 1.1 erläutert, sind zur Beschreibung und zum Austausch von Informationen zum Zusammenbau mehr Daten notwendig als CAD-Programme bieten können. Die Einbeziehung von abstraktem Wissen des Konstrukteurs in einer einheitlichen, maschinenlesbaren Form - einem „semantischen Modell“ - ermöglicht besseres Produktdesign und erleichtert die Zusammenarbeit und Wiederverwendung. Werden die Anforderungen und

Gründe für Designentscheidungen zusammen mit der Geometrie modelliert, so erlaubt dies auch eine automatisierte Analyse der Modellkonsistenz und Vergleiche mit anderen Modellen. Die gesammelten Informationen werden damit innerhalb einer Firma einfacher mehrfach nutzbar [6].

Die Schwierigkeit bei der Zusammenarbeit und beim Vergleich von Modellen liegt in der Verwendung heterogener Werkzeuge und verschiedener Vokabulare, die sich von Firma zu Firma oder gar von Konstrukteur zu Konstrukteur unterscheiden können. Werkzeuge legen Daten oft in proprietären Formaten ab, die einen Austausch über Systemgrenzen hinweg verhindern oder erschweren. Konstrukteure verwenden ihre eigenen Begriffe und Definitionen zur Beschreibung ihrer Anforderungen und Designs [7].

Sollen die verschiedenen Begriffswelten zusammengebracht werden, erfordert dies ein System, das die Intentionen der Konstrukteure abbilden kann und die Bedeutung der verwendeten Begriffe in einer einheitlichen Weise interpretierbar macht. Erst wenn sich verschiedene Design-Modelle in einer einheitlichen „Begriffswelt“ befinden, sind diese wirklich miteinander vergleich- und kombinierbar.

Mit STEP („Standard for exchange of product model data“ dt.: Standard für den Austausch von Produktmodelldaten, [8]) existiert ein in der ISO10303 genormtes Format, das ein möglichst breites Spektrum an Produktmodelldaten austauschbar machen möchte. Allerdings bietet es nur limitierte Möglichkeiten zur Beschreibung von Zusammenbauten.

Das „Semantic Web“ (oder „Web of Data“) bietet Mechanismen an, Daten zu strukturieren und um semantische Informationen zu ergänzen. Dies soll implizite Informationen expliziter machen und sie in einen Kontext setzen, damit sie besser maschinenverarbeitbar werden. Ontologien dienen als Quelle für gemeinsame, wohldefinierte Begriffe um Ressourcen zu beschreiben und ihre Zugänglichkeit in automatisierten Prozessen zu verbessern. Sie definieren den Kontext für den Datenaustausch.

1.3 Ziele

Ziel dieser Arbeit ist die Entwicklung einer Ontologie zur Beschreibung des Zusammenbaus von Produkten aus mehreren Teilen. Dabei wird der Fokus vor allem auf die Aufbaustruktur und Reihenfolge der beteiligten Komponenten gelegt. Dies soll verschiedene Einsatzzwecke bedienen, wie etwa den automatisierte Zusammenbau von Baugruppen oder die Generierung von Aufbauanleitungen zu unterstützen, kann aber auch im umgekehrten Sinne verwendet werden, um beispielsweise Zerlegungen von gebrauchten Produkten für Recyclingprozesse Informationen zu geben. Durch die Ontologie sollen diese Zusatzinformationen, die meist nicht in den CAD-Daten vorhanden sind, maschinenlesbar und leicht austauschbar gemacht werden. Beigefügte Regeln sollen die Nutzung der Ontologie vereinfachen, da damit aus den vorhandenen Daten zusätzliches Wissen generiert werden kann.

Um die Funktion der Ontologie und des Regelwerkes verifizieren zu können, wird eine Beispielimplementierung für den Import von CAD-Daten in die Ontologie demonstriert.

Dazu werden Zusammenbaudaten für Lego®-Modelle ausgelesen und entsprechend Teile- und Baugruppenbeschreibungen, Bauteilgruppen und Arbeitsschritte erzeugt.

Eine einfache Java-Applikation soll Benutzerinnen und Benutzern ermöglichen, das Regelwerk auf einfache Weise auch ohne Entwicklungsumgebungen zu benutzen.

1.4 Beitrag zu aktuellen Technologien

Für die Herstellung von Produkten, die aus mehren Teilen zusammengebaut werden müssen, sind oftmals spezialisierte Fertigungsanlagen anzutreffen. Wird die Produktion eingestellt, müssen diese Anlagen ebenso abgebaut werden, da sie nur an die Herstellung spezieller Produkte angepasst sind [9]. Das Wissen, das für die Durchführung des Zusammenbaus notwendig ist, steckt bei solchen Anlagen implizit in deren Aufbau. Ein Beispiel dafür sind Führungsschienen über die zwei Bauteile zueinander positioniert werden. Die Information, wie diese Teile zueinander positioniert werden müssen, steckt „hardcodiert“ in der Position der Führungsschienen.

Flexible Fertigungsanlagen spielen im momentanen Trend der „Industrie 4.0“ eine zentrale Rolle [10]. Sie benötigen das Wissen über den Zusammenbau explizit in maschinenlesbarer Form. Beispielsweise benötigt ein Bestückungsroboter Bauteilpositionen. Sollen diese nicht fix ins Roboterprogramm codiert sein - was ein Widerspruch zur geforderten Flexibilität ist - so müssen sie in einer geeigneten Form für den Roboter lesbar vorhanden sein, etwa in einer Datenbank. Ein generisches Programm kann diese Informationen zur Bewegungssteuerung des Roboters nutzen.

Diese Arbeit stellt eine Möglichkeit vor, Wissen über den Zusammenbau in einer Form abzulegen, die sowohl von Maschinen interpretiert werden kann, als auch von Menschen einfach verarbeitet werden kann. Dieses Wissen entsteht beim Design der Produkte und lässt sich, wie in der Einführung erläutert, oft nur unvollständig in CAD-Daten ablegen. Die Verwendung einer Ontologie bietet zusätzliche Vorteile:

- Ontologien enthalten Informationen in einer Form, die es ermöglicht daraus über Regeln und logisches Schließen neue Informationen zu generieren und damit implizites Wissen explizit zu machen. Dies wird in dieser Arbeit beispielsweise genutzt um aus einer Bauteilhierarchie einen Plan für den Zusammenbau abzuleiten.
- Ontologien sind offen gestaltet, d.h. darin enthaltene Informationen lassen sich ohne Eingriffe in die bestehenden Strukturen erweitern und mit anderen semantischen Daten verknüpfen.

In der gezeigten Ontologie kann der logischen Aufbau eines mehrteiligen Produktes abgebildet werden. Zusätzlich zu CAD-Daten in denen die Anordnung von Bauteilen im fertigen Produkt schon enthalten ist, besteht so die Möglichkeit die Reihenfolge der Fügungen zu beschreiben und Abhängigkeiten zwischen einzelnen Schritten des

Zusammenbaus festzulegen. Mehrfach verwendete Bauteile und Baugruppen können in einer Beschreibung zusammengefasst und mehrfach verwendet werden.

Das beigefügte Regelwerk kann aus diesen Bauteilstrukturen einen Plan generieren, aus dem eine flexible Fertigungsanlage herauslesen kann,

- in welcher Reihenfolge Fügungen zwischen Bauteilen passieren müssen, um ein fertiges Bauteil zu erhalten.
- bei welchen Fügungen Verzögerungen - ob gewollt oder ungewollt - tolerierbar sind, ohne den gesamten Zusammenbauplan zu gefährden.
- welche Schritte des Zusammenbaus parallelisierbar sind, und in welchem Zeitrahmen diese durchgeführt werden müssen.

Eine autonome Fertigungsanlage mit multiplen Fertigungsstationen kann mit diesen Information Arbeitsschritte auf ihre Stationen verteilen, sodass eine möglichst optimale Auslastung erreicht wird.

Die Ontologie sieht zusätzlich Möglichkeiten vor, Bauteilgeometrien, die für den Zusammenbau relevant sind, einzubetten. Aus diesen können grundlegende Merkmale wie Kanten, Flächen und Punkte abgeleitet werden. Damit lassen sich auch Beziehungen wie Parallelität oder Kollinearität bestimmen. Diese können dann zusammen mit den Informationen zur Ausrichtung der Bauteile in einem gemeinsamen Koordinatensystem bei der Fertigung für die Positionierung Vorteile bringen. Beispielsweise könnten Roboterbewegungen damit geplant werden.

Auch für eine (teil-) automatisierte Erstellung von Anleitungen sind diese Informationen wertvoll sein, da damit gezielt wichtige Bauteilbeziehungen betont werden können, damit diese in der Anleitung besonders berücksichtigt werden (z.B. in Beschreibungstexten wie „richten Sie Kante a parallel zu Bolzen b aus“ oder entsprechenden Grafiken).

1.5 Aufbau dieses Dokuments

Kapitel 2 führt in die semantischen Technologien ein und zeigt einige exemplarische Ontologien, die im Bereich des Assemblings bereits existieren.

Kapitel 3 erläutert die erstellte Zusammenbau-Ontologie und ihren Aufbau und gibt einen Überblick über die zugehörigen Regeln.

Kapitel 4 bietet einen detaillierteren Blick auf Kernpunkte der Umsetzung, die dabei verwendeten Werkzeuge und ausgewählte Teile des Regelwerkes.

Kapitel 5 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche Erweiterungen.

KAPITEL 2

Ontologien und das Semantic Web

> „The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.“ [11]
> (dt.: „Das Semantic Web ist eine Erweiterung des herkömmlichen World Wide Web (WWW), in der Informationen mit eindeutigen Bedeutungen versehen werden, um die Arbeit zwischen Mensch und Maschine zu erleichtern.“)

Das Konzept des „Semantic Web“ geht zurück auf einen Vorschlag von Sir Tim Berners-Lee, dem Begründer des World Wide Web. Die Vision war es für das WWW ein gemeinsames Framework zu schaffen, das es ermöglicht zusätzliche, maschinenlesbare Information über alle Firmen und Applikationsgrenzen hinaus zu teilen. Maschinen sollten dazu befähigt werden aus den bisher unstrukturierten Daten Schlussfolgerungen zu ziehen [11].

Die Basis des Semantic Web sind Uniform Ressource Identifiers (URIs), die Ressourcen und Begriffe eines Datenbestandes bzw. Dokumentes im Netz mit anderen Ressourcen und Begriffen im Netz verbinden. Inhalte, die in einem Dokument beschrieben werden, sind somit nicht nur im Text enthalten, sondern durch eindeutige Verknüpfungen mit Begriffen und anderen Ressourcen näher beschrieben. Diese Struktur von Verlinkungen erlaubt es Computerprogrammen nun, Fakten aus einem Dokument abzuleiten, auch wenn diese vom Autor des Dokumentes gar nicht erzeugt wurden. Die Ursprünge kommen aus dem Forschungsbereich der Wissensrepräsentation, einem Teilbereich der künstlichen Intelligenz.

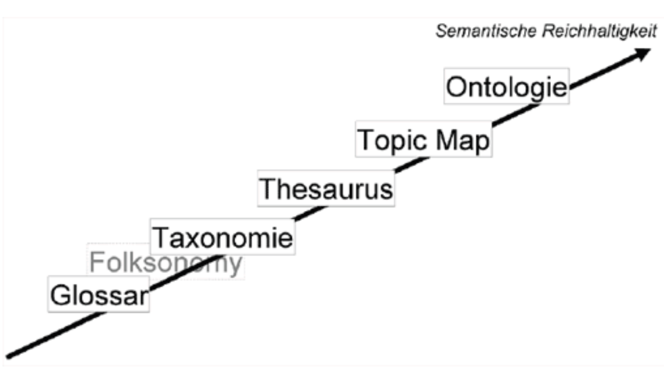


Abbildung 2.1: „Semantische Treppe“ [12]

2.1 Semantische Ansätze

Im Folgenden werden einige Begriffe aus dem Bereich der Wissensrepräsentation und semantische Ansätze erläutert. Die Reihenfolge ergibt sich aus der steigenden Komplexität und Ausdrucksfähigkeit der Modelle (Abbildung 2.1). Da mehrere Definitionen in der Literatur existieren, wird hier bewusst auf einige wenige Quellen referenziert um eine durchgängige Linie zu behalten.

2.1.1 Glossar

Ein Glossar ist eine Liste von Begriffen und einer zugehörigen Erklärung. Beziehungen zwischen den Begriffen können formal nicht festgelegt werden.

2.1.2 Folksonomie

In einer Folksonomie werden Inhalten Schlagworte (Tags) von sehr vielen Benutzerinnen und Benutzern ohne Regeln zugeordnet. Die Sammlung dieser Tags wird Folksonomie genannt. Diese Art der Wissensanreicherung ist vor allem im Zusammenhang mit sozialen Netzwerken verbreitet. Auch zwischen Tags gibt es keine formalen Beziehungen, allerdings werden Ressourcen, die mit gleichen Tags versehen sind, oft als in Beziehung stehend betrachtet.

2.1.3 Taxonomie

Eine Taxonomie bezeichnet eine Hierarchie von Begriffen Abbildung 2.2. Taxonomien werden in verschiedenen Wissenschaftsdisziplinen verwendet. Anwendung finden sie zum Beispiel in der Ordnerstruktur von Dateisystemen. Die Pfeile in der Grafik haben keinen eindeutigen Bedeutungsinhalt. Soll ein Begriff mehreren Oberbegriffen zugeordnet werden, muss er mehrfach in der Taxonomie angelegt werden.

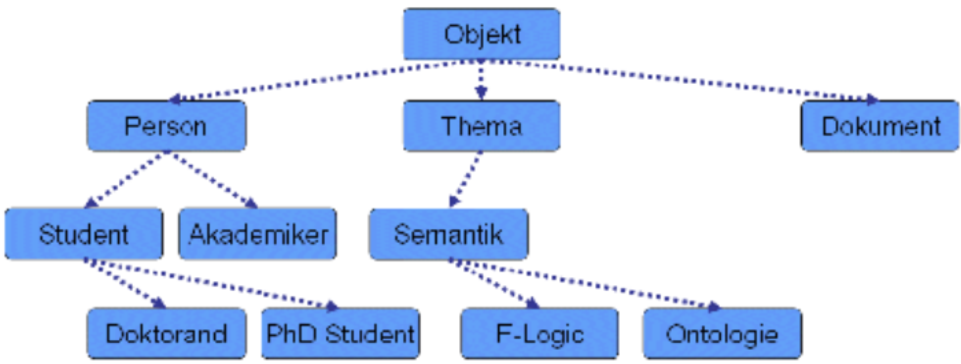


Abbildung 2.2: Beispiel einer Taxonomie [13]

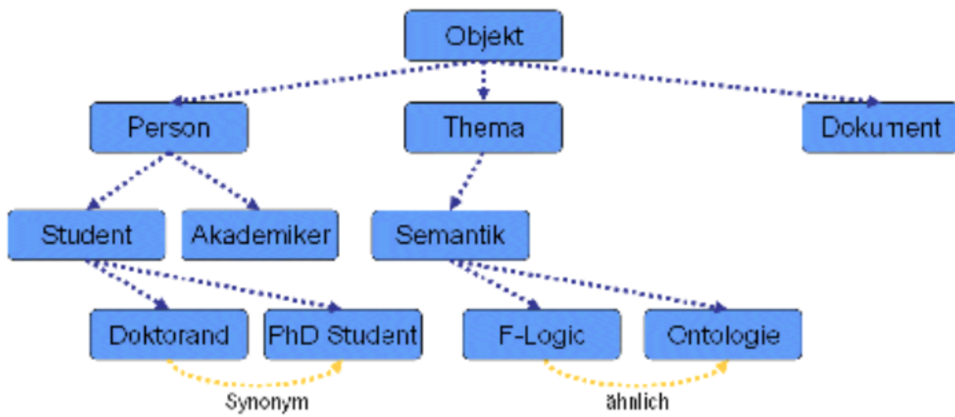


Abbildung 2.3: Beispiel eines Thesaurus [13]

2.1.4 Thesaurus

Thesauren kommen aus dem Bereich der Bibliotheken. Sie erweitern die Hierarchie einer Taxonomie um zwei Beziehungsmöglichkeiten: Synonym und Ähnlichkeitsrelation. Weitere eigene Relationen sind nicht möglich. Abbildung 2.3 zeigt, wie diese Beziehungstypen die Information im Beispiel erweitern um z.B. auszudrücken, das Doktorand und PhD-Student äquivalent sind.

2.1.5 Topic Map

Topic Maps sind als ISO-Standard auf Basis der eXtensible Markup Language (XML) (früher Standard Generalized Markup Language (SGML)) spezifiziert [14]. Sie bestehen aus abstrakten Dingen (Subjects), deren Repräsentationen (Topics), Assoziationen, Gültigkeitsbereichen von Topics (Scopes) und zugeordneten Dokumenten außerhalb der Topic Map (Occurences). Im Standard ist nur der Aufbau der Topic Map beschrieben,

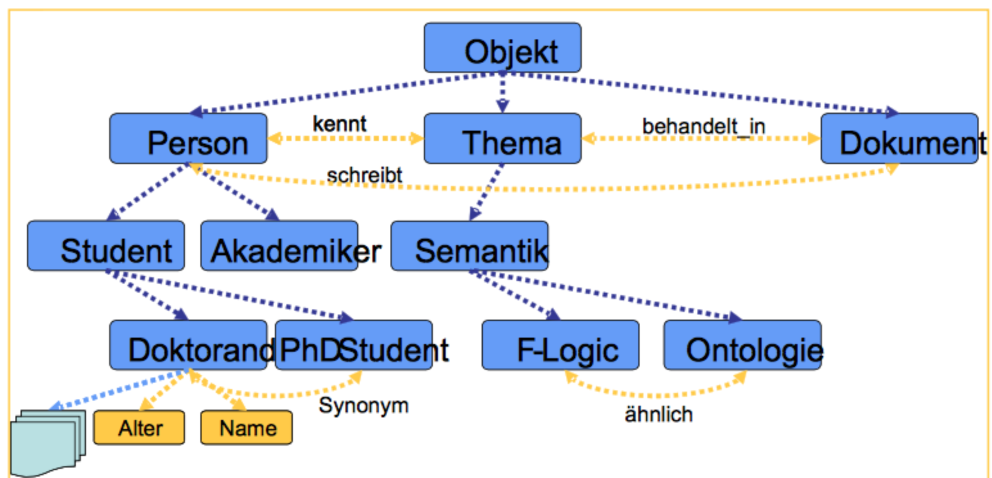


Abbildung 2.4: Beispiel einer Topic Map [13]

eine standardisierte Abfragemöglichkeit ist zu diesem Zeitpunkt erst in Begutachtung [15]. Angewendet auf das Beispiel aus den vorigen Abbildungen ergeben sich (vereinfacht dargestellt) die in Abbildung 2.4 gezeigten Erweiterungen.

Es ist jetzt möglich, speziellere ungerichtete Beziehungen zwischen den einzelnen Begriffen selbst herzustellen (z.B. eine Person schreibt ein Dokument) und Topics mit Attributen zu versehen (z.B. Alter). Topic Maps verwenden bereits URIs als Identifikationsmerkmale.

2.1.6 Ontologien

Der Begriff Ontologie wurde in der Philosophie als „Lehre des Seins“ geprägt. Im informatischen Kontext wird oft auf die Definition von Thomas Gruber verwiesen: „An ontology is a specification of a conceptualization.“ [16] (dt.: „Eine Ontologie ist eine explizite Spezifikation einer (gemeinsamen) Konzeptionalisierung.“) Ontologien erlauben Konzepte einer Domäne und Beziehungen zwischen diesen in einer bedeutsamen Weise zu beschreiben.

Als Basis dienen Datenmodelle, die im Ressource Description Framework (RDF) formuliert werden. Diese definieren Ressourcen und deren Beziehungen zueinander innerhalb einer Anwendungsdomäne [17].

Die Semantik und einfache Abhängigkeiten lassen sich mit dem Ontologie-Vokabular unter Verwendung von RDF Schema (RDFS) festlegen. Die OWL Web Ontology Language (OWL) erweitert diese Möglichkeiten noch. So kann damit beispielsweise zwischen Klassen (Konzepten) und ihren Instanzen (Anwendung der Konzepte) unterschieden werden oder Instanzen benannte Eigenschaften zugeordnet werden ([18],[19]).

Komplexere logische Zusammenhänge können über Regeln ausgedrückt werden. Dafür existieren verschiedene regelbasierte Sprachen wie Semantic Web Rule Language (SWRL)

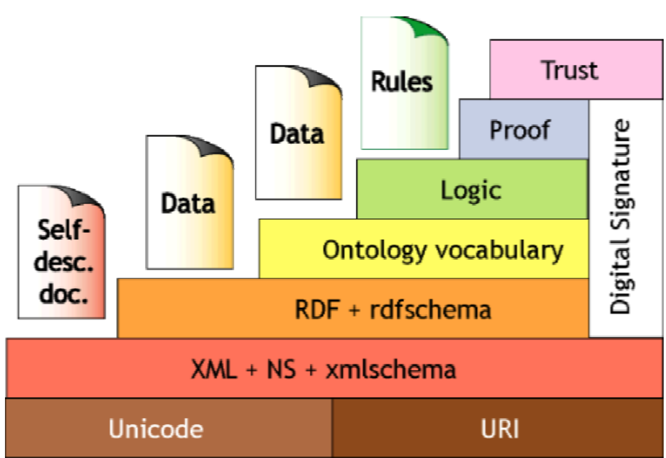


Abbildung 2.5: Schichten des Semantic Web nach Tim Berners-Lee [13]

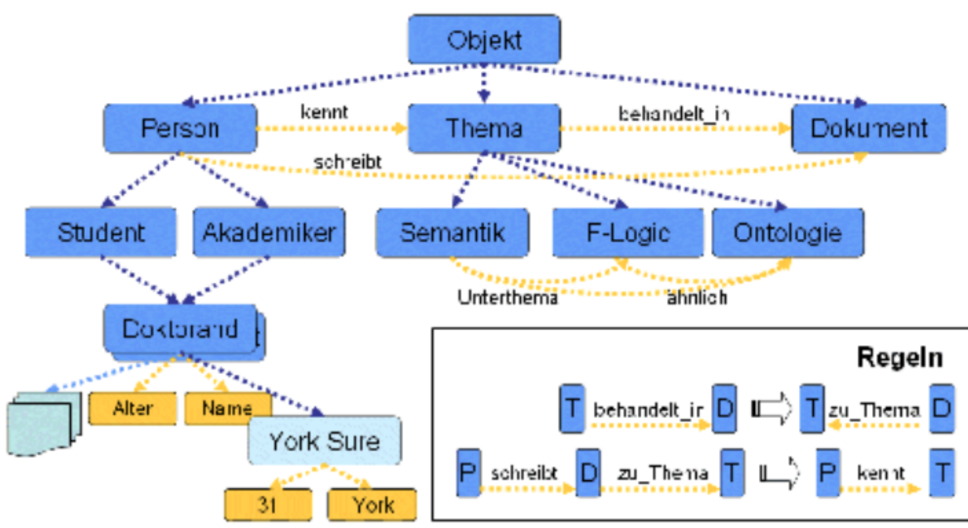


Abbildung 2.6: Beispiel einer Ontologie [13]

oder die in dieser Arbeit verwendete Jess®, the Rule Engine for the Java Platform (Jess).

Abbildung 2.5 veranschaulicht den Zusammenhang dieser Komponenten einer informatischen Ontologie.

Im in Abbildung 2.6 gezeigten Beispiel ist die Domäne eine Universität mit Konzepten wie etwa „Thema“, „Person“ oder „Dokument“ und dazugehörigen Beziehungen wie „schreibt“ oder „kennt“. Es kann jetzt zwischen „is-a“-Beziehungen (Vererbung, „ein Student ist eine Person“) und „has-part“-Beziehungen („eine Person schreibt ein Dokument“) unterschieden werden. Schema und Inhalte werden klar voneinander getrennt. Konzepte definieren das Schema, während Individuen (Instanzen) die jeweiligen Ausprägungen

darstellen. So ist im Beispiel „York Sure“ eine Instanz des Konzepts „Doktorand“.

Regelwerke legen Zusammenhänge über Wenn-Dann Beziehungen, Zuweisungen, logische Verknüpfungen, etc. genauer fest. Im Beispiel sind die Beziehungen „Person schreibt Dokument“ und „Thema behandelt_in Dokument“ definiert. Eine einfache Regel kann eine Inversität definieren, sodass, wenn ein Thema in einem Dokument behandelt wird, dieses Dokument ein Thema („Dokument zu_Thema Thema“) beschreibt. Eine komplexere Regel könnte lauten: „wenn eine Person ein Dokument zu einem Thema verfasst, kennt sie dieses Thema“. Hier zeigt sich eine der Stärken einer Ontologie. Es lassen sich anhand logischer Ableitung der Regeln Fakten ermitteln, die nicht explizit definiert wurden. Der Umstand dass eine Person ein Thema kennt, kann aus der Information, dass sie eine Arbeit schreibt und den beiden exemplarischen Regeln geschlussfolgert werden.

Nach [13] eignen sich Ontologien ebenso wie Topic Maps zur Erweiterung von Suchsystemen oder zur Verknüpfung heterogener Datenquellen.

Open World Assumption

Ontologien bilden immer einen Ausschnitt der realen Welt bzw. ihrer Domäne ab. Anders als in relationalen Datenbanken, bei denen nur Inhalte gespeichert werden können, die in der Struktur vorgesehen sind (Closed World Assumption (CWA)), präsentieren sich Ontologien hier offener. Semantische Websprachen gehen von der Open World Assumption (OWA) aus, d.h. es wird angenommen, dass die vorhandenen Daten nicht vollständig sind um eine Aussage zu bestätigen oder zu widerlegen [20].

Tier	kann fliegen
Taube	ja
Hai	nein
Pinguin	nein

Tabelle 2.1: Daten für ein Beispiel zu OWA(nach [20])

Die Frage „Können Schweine fliegen“ unter Berücksichtigung der Daten in Tabelle 2.1 würde in der CWA - also beispielsweise bei einer Structured Query Language (SQL)-Abfrage mit nein beantwortet werden, da keine Daten dazu vorliegen. In der OWA würde die Antwort „Unbekannt“ lauten, solange es kein Statement wie „Schweine können nicht fliegen“ gibt, oder dies aus anderen Statements logisch abgeleitet werden kann.

Ontologien eigenen sich also ausgezeichnet dafür, Wissen zu einem bestimmten Anwendungsbereich maschinenlesbar abzulegen und zu repräsentieren. Dies ergibt sich aus folgenden Eigenschaften (vgl. auch vorangegangene Abschnitte):

- Wortschätze können frei definiert und Beziehungen zwischen den einzelnen Vokabeln hergestellt werden.
- Die Verwendung der Vokabeln kann durch Regeln näher spezifiziert werden.

- Ontologien nach den Mechanismen des Semantic Web basieren auf offenen Standards, sind also nicht an Hersteller oder Systeme gebunden.
- Die enge Verknüpfung mit Webtechnologien bringt inhärente Möglichkeiten zum Austausch und zur Zusammenarbeit mit sich.

2.2 Entwicklung einer Ontologie

> „1) There is no one correct way to model a domain — there are always viable alternatives. The best solution almost always depends on the application that you have in mind and the extensions that you anticipate.
> 2) Ontology development is necessarily an iterative process." [21]
> (dt.: „1) Es gibt nicht nur einen korrekten Weg, um eine Domäne zu modellieren - es gibt immer sinnvolle Alternativen. Die beste Lösung hängt fast immer vom geplanten Anwendungsfall und den erwarteten Erweiterungen ab.
> 2) Ontologie-Entwicklung ist zwangsläufig ein iterativer Prozess.")

Laut [21] ist Ontologie-Entwicklung somit ein iterativer Prozess, der sich in mehreren Zyklen einem verwendbaren Ergebnis annähern wird.

2.2.1 Domänenanalyse

Im Zuge der Domänenanalyse sind folgende Fragestellungen zu bearbeiten [21]:

- Welche Domäne soll die Ontologie abdecken?
- Wofür soll die Ontologie verwendet werden?
- Welche Art von Fragen sollen sich durch das in der Ontologie abgebildete Wissen beantworten lassen? Werden schon zu Beginn Kompetenzfragen [22] zur Anwendungsdomäne formuliert, stellen diese eine Test-Möglichkeit für die Fähigkeiten der erstellten Ontologie dar.
- Wer wird diese Ontologie nutzen?

Die Antworten auf diese Fragen können sich auch während der Design-Phase ändern, sie helfen aber dabei den Umfang des Modells klar einzugrenzen.

Die Wiederverwendung bestehender Ontologien zur abzubildenden Domäne ist in zweierlei Hinsicht sinnvoll. Erstens erspart es den Aufwand der eigenständigen Erstellung aller Konzepte der Domäne, vielmehr kann auf den vorhandenen aufgebaut und eventuell fehlende für das eigene Ziel ergänzt werden. Zweitens kann dies gleich ein Anknüpfungspunkt für die Zusammenarbeit mit Datenquellen sein, die das eigene Modell um bisher nicht angedachte Informationen ergänzen.

2.2.2 Konzeptionalisierung und Implementierung

Die weitere Entwicklung kann in Schritte aufgeteilt werden:

- Auflistung aller für den späteren Einsatz der Ontologie relevanten Konzepte (Klassen) in der Domäne
- Verortung dieser Konzepte in einer Taxonomie
- Definition von Attributen für die Klassen und deren Wertebereich
- Erzeugung von Instanzen mit konkreten Werten in deren Attributen

Die daraus gewonnen Daten müssen in einer Ontologiesprache (z.B. OWL) implementiert werden.

2.2.3 Test und Wartung

Die erwähnten Kompetenzfragen - idealerweise unter Einbeziehung von Experten der Domäne - dienen zur Überprüfung der Tauglichkeit einer Domäne für den geplanten Zweck. Zusätzlich müssen aber auch formale Evaluationen, wie Konsistenzprüfungen mit einer Software zum automatischen Ziehen von logischen Schlüssen („Reasoner“), durchgeführt werden.

Die dabei oder auch bei der tatsächlichen Verwendung aufkommenden Änderungs- und Verbesserungsvorschläge werden daraufhin in einem weiteren Zyklus der Entwicklung eingearbeitet. Der Grad der Änderungen sollte natürlich mit jedem neuen Durchlauf geringer werden, andernfalls würde man sich von einer optimalen Lösung entfernen.

2.3 Bisherige Lösungsansätze

Dieser Abschnitt zeigt verschiedene bisherige Lösungsansätze um den Zusammenbau von Produkten über semantische Technologien zu beschreiben.

Ontologien existieren für eine Vielfalt von Anwendungsdomänen, oftmals in einer sehr umfassenden Form. In Top-Level Ontologien werden Daten auf einem sehr allgemeinen Level gesammelt und erlauben logisches Schlussfolgern über Konzepte und Sachverhalte, die domänenübergreifend gültig sind. Ein Beispiel ist CYC [23], eine kommerzielle von der Firma Cycorp verwaltete Ontologie mit mehreren Millionen Fakten und einfachen Regeln zu Alltagswissen. Mögliche Anwendungsgebiete liegen beispielsweise in der Verbesserung der maschinellen Verarbeitung von natürlicher Sprache (in der Hauptsache Englisch) bei Spracherkennung und maschineller Übersetzung oder bei der Filterung und Priorisierung von textuellen Kommunikationsdaten wie eMails. Auch wenn CYC nicht unmittelbar in der Domäne mechanischen Designs und Produktionsprozessen angesiedelt ist, wurde es doch vom amerikanischen National Institute of Standards and Technology (NIST)

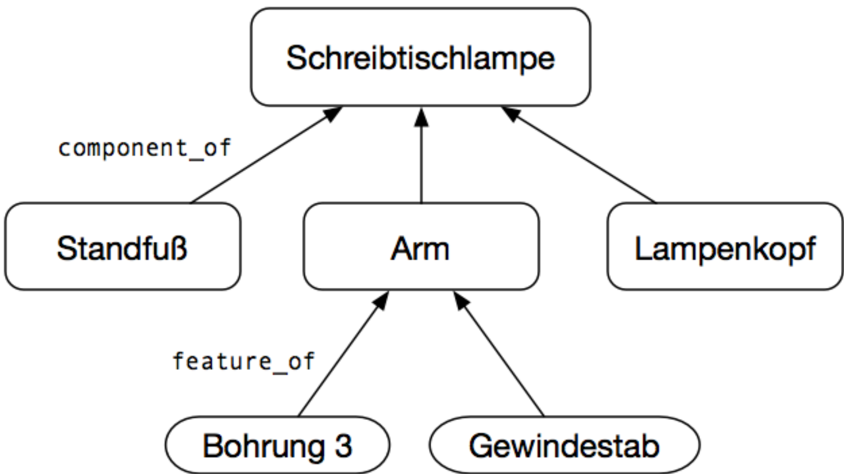


Abbildung 2.7: Teil der Ontologie nach [7] zur Beschreibung eines Produktes

für weitere Untersuchungen zum Einsatz in diesen herangezogen [24]. Die Ergebnisse führten zur Entwicklung der Process Specification Language (PSL), einer Sprache, die Beschreibungen von Konstruktions- und Fertigungsprozessen austauschbar macht [25].

Deutlich spezifischer für die Engineering-Domäne formuliert [7] eine Ontologie um Wissen um das Design von Produkten explizit zu machen. Die Ontologie beinhaltet eine Abbildung von Produkten und eine Hierarchie der beteiligten Bauteile („Parts"). Dabei wird zwischen primitiven (unteilbaren) und zusammengesetzten Teilen unterschieden. Jedem Bauteil können Parameter wie Farbe oder Gewicht sowie geometrische und funktionale Merkmale („Features") zugeordnet werden (Abbildung 2.7).

Das Hauptaugenmerk der Ontologie ist die Modellierung von Anforderungen für Produkte, daher sind Beschreibungen der Produkte selbst auf die gezeigten Möglichkeiten eingeschränkt. Weitere spezifischere Merkmale für den Zusammenbau oder Informationen für die Fügung von Bauteilen sind nicht vorgesehen.

Ein sehr detaillierter Ansatz für das Design von Fügungen ist bei Kim et al. [26] zu finden. Ihre Motivation für die nähere Beschreibung von Verbindungen zwischen Teilen ist es, kollaboratives Konstruieren zu unterstützen und die Beschreibungen (im Text als AsD - AssemblyDescription bezeichnet) leicht austauschbar zu machen:

> „AsD participants, such as customers, suppliers, assembly designers, production engineers, and other stakeholders, need to exchange AsD information seamlessly in a collaborative environment." [26] (dt. „AsD Teilnehmer wie Kunden, Zulieferer, Konstrukteure, Fertigungstechniker und andere Akteure, müssen AsD-Informationen in einer gemeinschaftlichen Umgebung nahtlos austauschen können.").

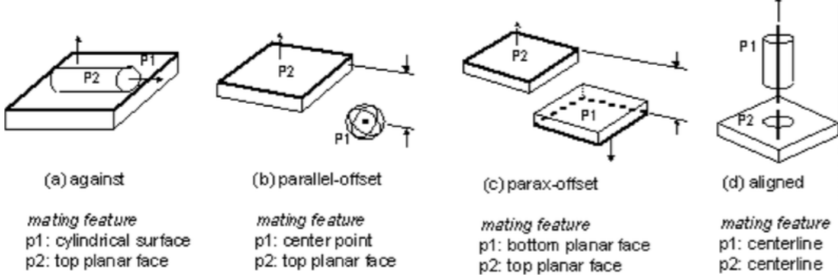


Abbildung 2.8: Räumliche Abhängigkeiten („Spatial Relationships") und deren Verbindungsmerkmale („Mating Features") nach [26]

Kim et al. legen das Hauptaugenmerk auf geometrische und räumliche Abhängigkeiten der an einer Verbindung beteiligten Teile zueinander. Eine Designerin oder ein Designer legt die räumlichen Beziehungen für zu fügende Bauteile fest. Daraus können die verbleibenden Freiheitsgrade der Bauteile abgeleitet werden. In einem zweiten Schritt werden dann alle an der Fügung beteiligten Merkmale, genannt „Mating Features", extrahiert (Abbildung 2.8). Die Autoren verwenden ebenso wie [7] den Begriff „Feature" für Merkmale des Produktes bzw. der Bauteile. Mating Features sind definiert als Satz von geometrischen Größen wie Flächen, Kanten oder Punkte, welche für die Angabe der relativen Position der zu fügenden Bauteile notwendig sind. Diese werden in den „Joint Features" (dt. Verbindungsmerkmale) mit Informationen und Parametern zum Verbindungsverfahren verknüpft. Letztlich entsteht eine umfassende Beschreibung einer Verbindung zusätzlich zu den CAD-Daten, die in einem kollaborativen Design einfach ausgetauscht werden können. Kim et al. verwenden ein eigenes XML-Format für den Datenaustausch und Beschreibung der AsD.

In der Arbeit von Manlay [27] wird das Modell von Kim et al. in eine OWL-Ontologie überführt (Abbildung 2.9). Regeln, formuliert in der Regelsprache SWRL, bilden die Feature-Extraktion und Ableitungen der Freiheitsgrade nach. Damit werden die AsD nach Kim et al. unmittelbar im Kontext des Semantic Web nutzbar und können mit weiteren Ontologien aus der Anwendungsdomäne des Produktdesigns verknüpft werden.

Die Modelle von Kim et al. und Manley sind für den Designprozess von mehrteiligen Produkten konzipiert. Designer können ihre CAD-Modelle um (in CAD-Tools) normalerweise nicht sichtbaren Verbindungsinformationen ergänzen. Bei der tatsächlichen Durchführung des Zusammenbaus in der Produktion spielen aber noch weitere Aspekte eine Rolle, die sich nicht einfach aus den CAD-Daten oder der AsD ableiten lassen, wie beispielsweise die Reihenfolge der Fügungen.

Liang und Paredis [28] abstrahieren die notwendigen Zusammenhänge zwischen Komponenten als sogenannte „Ports". Ein Port ist definiert als „location of intended interaction" [28] (dt. Stelle der gewollten Interaktion) zwischen zwei Komponenten oder Subsystemen (Abbildung 2.10). Dieser Ansatz lässt sich interdisziplinär nutzen, sowohl für das Produktdesign als auch für Simulationsanwendungen, da ein Port vieles darstellen kann, wie etwa einen elektrischen Anschluss oder den Zapfen eines Lego®-Steins. Aus der Ontologie

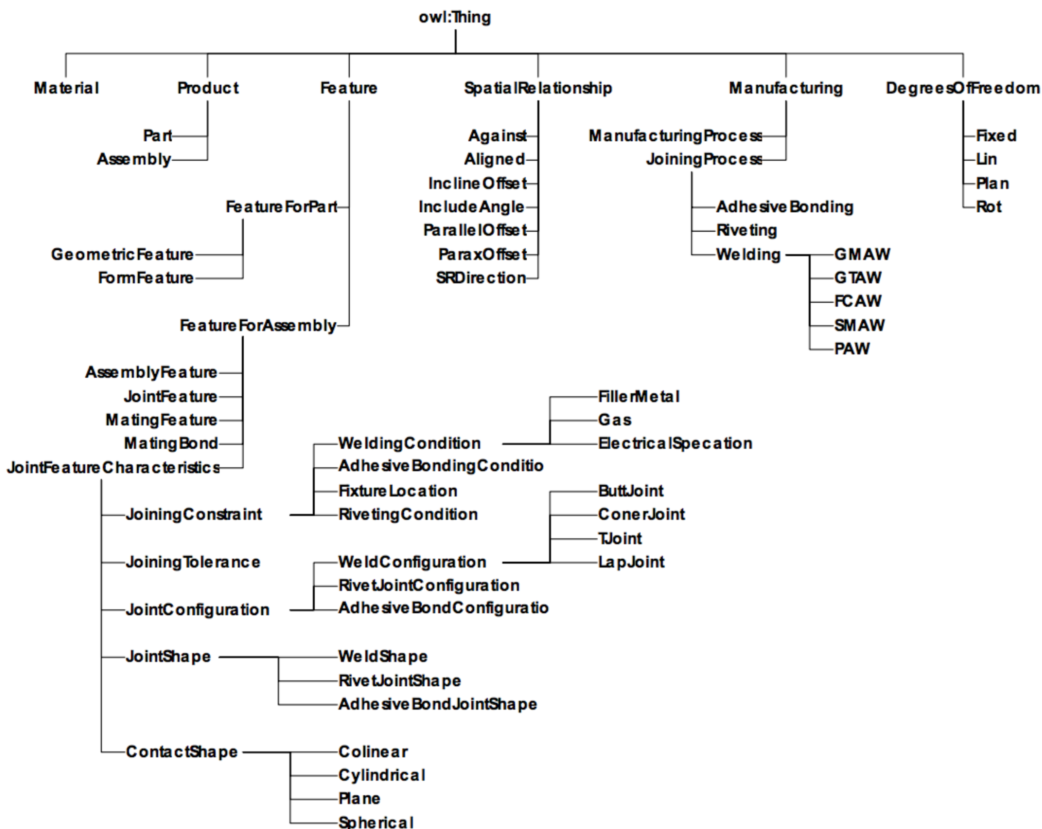


Abbildung 2.9: Klassen der AsD-Ontologie nach [27]

lässt sich die Kompatibilität der Ports untereinander und Eigenschaften der Verbindung herauslesen. Liang und Paredis beantworten mit ihrer Ontologie mehr die Frage nach dem „Warum“ einer Verbindung, beispielsweise über eine Verbindungseigenschaft “conveyEnergy"(dt. transportiert Energie). Wie diese Verbindung zu Stande kommt ist nicht Teil des Modells.

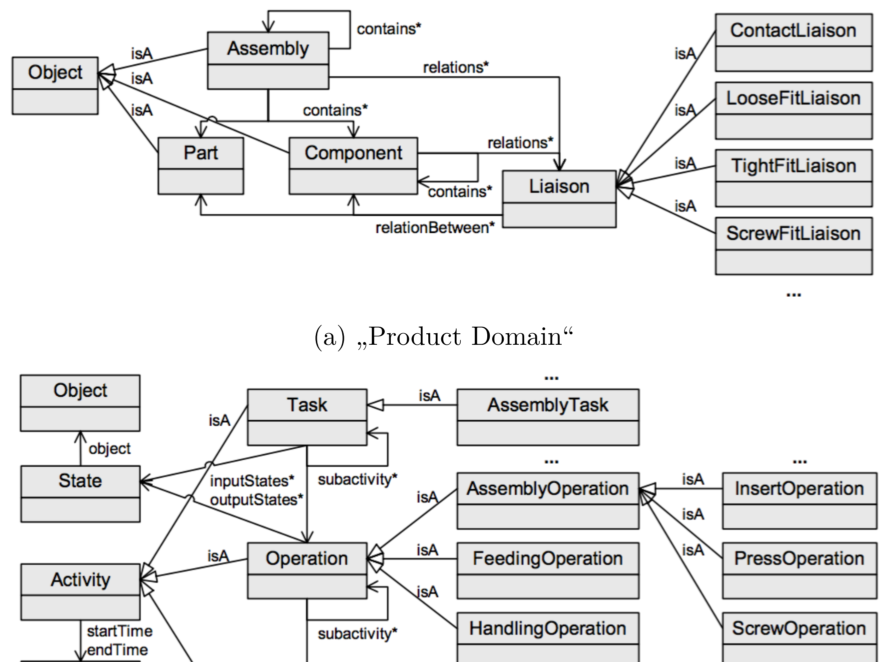
Eine Ontologie zur Beschreibung des Zusammenbau-Prozesses für die Unterstützung flexibler Fertigungen findet sich bei Lohse et al. [9]. Diese bietet der Benutzerin oder dem Benutzer die Möglichkeit, die Vorgehensweise beim Zusammenfügen von Bauteilen in verschiedenen Detailstufen zu spezifizieren. Der räumliche Zusammenhang zwischen Bauteilen (Teile ohne interne Freiheitsgrade), Komponenten (Teilen mit mindestens einem internen Freiheitsgrad) und Baugruppen wird über sogenannte „Liaisons“ (dt. Verbindungen) hergestellt, die in etwa den „Spatial Relationships“ bei Kim et al. [26] entsprechen (Abbildung 2.11 (a)). Zusammenbauprozesse werden in „Assembly-Tasks“ beschrieben (Abbildung 2.11 (b)). Diese werden auf die notwendigen Operationen heruntergebrochen. Diese Operationen beziehen sich anders als bei [26] auf die Positionierung der Bauteile zueinander. Dies inkludiert auch komplexe Vorgänge wie das Zurückdrücken

Abbildung 2.10: Beziehungen zwischen Komponenten mit „Ports“ nach [28]

einer Feder um die endgültige Lage der beteiligten Komponenten zu erreichen. Sind die Operationen identifiziert können daraus Aktionen abgeleitet und näher spezifiziert werden. Aktionen beschreiben konkrete Handlungen wie „fixieren“ oder „bewegen“. Die Ontologie von Lohse et al. ist sehr spezifisch für das Zusammenbringen der Komponenten. Die notwendigen Operationen werden nicht nur in ihrer Art, sondern auch in ihren Abhängigkeiten zueinander beschrieben. Somit kann aus der Ontologie auch ein Ablauf der Fügung gelesen werden.

Die Beschreibung des Zusammenbaus von Produkten ist aber nicht nur für das Produktdesign oder die Fertigung von Interesse. Viele Alltagsprodukte wie Möbel oder Spielzeuge erfordern den Zusammenbau durch Endkundinnen und Endkunden. Bei solchen Produkten wird eine Anleitung mitgeliefert, die den Zusammenbau - oftmals in Form von Bildern - beschreibt (Abbildung 2.12) [29]. Für modulare Produktlinien wie z.B. Büromöbel sind viele verschiedene Anleitungen notwendig. Mit dem steigenden Bedürfnis an individuellen Produkten wird auch der Bedarf an Anleitungen größer. Anleitungen zu erstellen, die leicht zu verstehen und nachzuvollziehen sind, ist schwierig und teuer, da hierfür noch viele manuelle Tätigkeiten von Grafik-Designerinnen und -Designern notwendig sind. Maschinelle Unterstützung kann dazu beitragen, Anleitungen kosteneffizienter zu gestalten [30].

Maneesh et al. [30] beschreiben die Informationen, die Leserinnen und Leser von Anleitungen erwarten bzw. die für die Gestaltung von Anleitungen wichtig sind. Dabei wird auch festgestellt, dass Benutzerinnen und Benutzer die Bauteile eines Produktes in einer hierarchischen Ordnung sehen, die sich aber nicht notwendigerweise mit einer funktionalen Baugruppen-Hierarchie decken muss. Die Verwendung einer Ontologie als Datenbasis erlaubt es sehr einfach, mehrere Hierarchien parallel aufzubauen. Dabei können die selben grundlegenden Informationen mehrmals verwendet und nur durch



(a) „Product Domain“

(b) „Assembly Process Domain“

Abbildung 2.11: Ausschnitte aus der Ontologie für die Beschreibung von Zusammenbauprozessen nach [9]

verschiedenartige Beziehungen untereinander neu verknüpft werden. Alternativ lassen sich auch Elemente, die in verschiedenen Teilen einer Ontologie oder gar in verschiedenen Ontologien gespeichert sind, gleichsetzen (beispielsweise über eine `sameAs`-Beziehung aus der OWL).

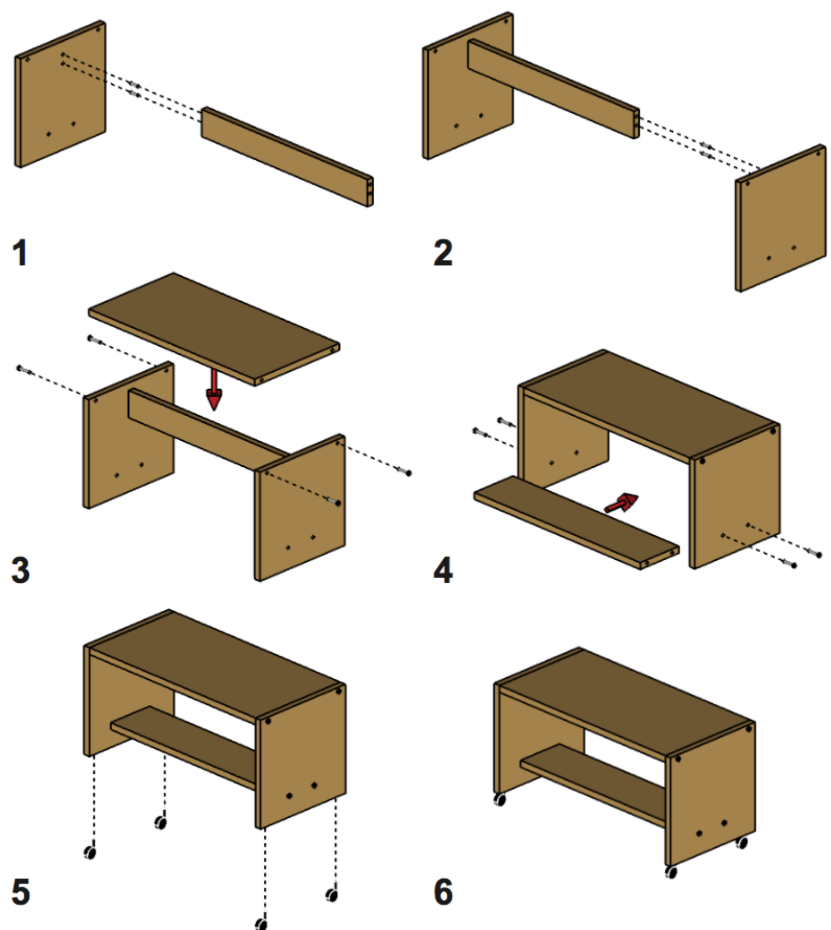


Abbildung 2.12: Zusammenbauanleitung eines TV-Tisches [30]

KAPITEL 3

Vom Produkt zur Aufbaubeschreibung

In diesem Kapitel wird das grundsätzliche Konzept der entwickelten Ontologie vorgestellt und das Zusammenspiel der einzelnen Teile beschrieben. Für eine Darstellung der modellierten Klassen bietet sich die Notation als Klassendiagramm in der Unified Modelling Language (UML) [31] an. Abbildung 3.1 zeigt einige Klassen der Ontologie als Rechtecke, deren Beziehungen als Verbindungslinien [1].

Ein Produkt in der Ontologie ist immer ein Individuum (eine Instanz) der Klasse `Product` (dt.: Produkt). Diese Klasse dient als Verknüpfungspunkt eines Produktes, dessen Zusammenbau beschrieben und geplant wird und anderen Ontologien, welche die hinterlegten Informationen verwenden. Beispielsweise beschreibt [32] ein wissensbasiertes Automatisierungssystem für flexible Fertigungsprozesse. Dieses könnte die hier beschriebene Ontologie dazu benutzen, um aus den geplanten Zusammenbauschritten eine Verteilung der Arbeiten auf verschiedene Fertigungsstationen abzuleiten und damit eine effiziente Fertigung zu planen.

Wie schon in Abschnitt 1.1 erläutert sind zur Beschreibung eines Zusammenbaus in einem Datenmodell grundlegend drei Fragestellungen interessant:

- *Wer?* Welche Bauteile sind bei einen Zusammenbauschritt beteiligt?
- *Wie?* Auf welche Weise werde die Bauteile zueinander gefügt, welche Aktionen und Operationen werden verwendet und gibt es hier Abhängigkeiten?

[1] In der hier vorgestellten Ontologie werden ausschließlich englische Begriffe für Klassen- und Attributnamen verwendet. Diese werden in weiterer Folge bei der ersten Erwähnung erläutert und übersetzt. In der weiteren Verwendung im Text werden anschließend nur noch die englischen Fachbegriffe in folgender Form gekennzeichnet verwendet z.B.: `SubAssembly`.

- *Wann?* In welcher Reihenfolge muss der Zusammenbau geschehen? Wie kann dieser geplant werden?

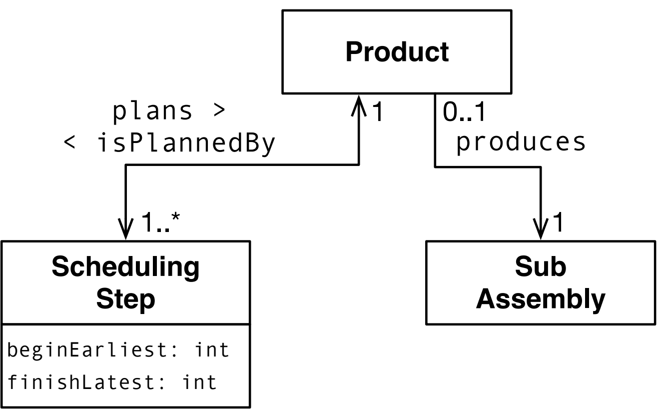


Abbildung 3.1: Darstellung von Klassen und Beziehungen zwischen Klassen in UML-Notation

pedes Produkt verweist auf ein `SubAssembly` (dt.: Baugruppe) und ein oder mehrere Individuen der Klasse `SchedulingStep` (dt.: Planungsschritt). Das referenzierte `SubAssembly` beschreibt den prinzipiellen Aufbau eines mehrteiligen Produkts und dient daher der Beantwortung der ersten Fragestellung. Im Detail wird dies in Abschnitt 3.1 erläutert. Abschnitt 3.2 beleuchtet die Teile der Ontologie, die sich mit dem „Wie?“, also mit den Aktionen, die für einzelnen Zusammenbauschritte notwendig sind, beschäftigen. Der letzten Fragestellung widmet sich Abschnitt 3.3. Es wird die Klasse `SchedulingStep` eingeführt, deren Individuen einen sequenziellen Ablauf zur Planung der Produkterstellung abbilden. Die Individuen der Klasse werden dabei über Inferenzregeln aus Daten über die Aufbaustruktur und die Verarbeitungsschritte abgeleitet.

3.1 Zusammenbaubare Teile und deren Beschreibungen

Wird ein mehrteiliges Produkt betrachtet, so kann dieses auf verschiedene Arten beschrieben werden:

- *Bauteilliste* Diese einfache Art der Beschreibung macht keinerlei Aussage, welche Bauteile miteinander in Beziehung stehen, sondern nur, welche Bauteile in welcher Anzahl im Produkt enthalten sind. Abbildung 3.2 zeigt diese Form am Beispiel von einigen Teilen eines Kraftfahrzeuges. Für Materialorganisation oder Bauteilebestellung mag das ausreichend sein. Da sich diese Arbeit mit dem Zusammenbau von mehrteiligen Produkten als Prozess beschäftigt, genügt diese Art der Beschreibung nicht, jedoch wird ein Regelprogramm vorgestellt, dass diese Aufzählungsform aus der Ontologie generieren und zurückliefern kann (siehe Abschnitt 4.5)

	Teil	Anzahl
	Reifen	4
	Felge	4
	Radmutter	24
	Kolben	4
	Pleuelstange	4
	Zylinderblock	4
	...	

Abbildung 3.2: Produktbeschreibung anhand einer (unvollständigen) Bauteilliste

- *Bauteilgruppen* Üblicherweise stehen die beteiligten Bauteile eines Produktes nicht gleichwertig und unabhängig nebeneinander, sondern es sind gewisse Abhängigkeiten vorhanden. Diese Abhängigkeiten können beispielsweise aus der Funktion der Bauteile entstehen: Felge, Reifen und Radmuttern eines Kraftfahrzeuges sind funktional zusammengehörig und könnten daher einer Baugruppe „Rad“ zugeordnet werden. Zylinder, Kolben und Pleuelstange könnten analog einer Baugruppe „Motor“ angehören. Die Baugruppen bilden also in der Produktbeschreibung eine zusätzliche Informationsebene (Abbildung 3.3). Können Baugruppen Unter-Baugruppen enthalten, entsteht eine Hierarchie, die sich über die Funktionalität der Elemente definiert.

- *Logische Bauteil-Hierarchie* Bauteile können auch aufgrund logischer Zusammenhänge modelliert werden. Beispielsweise ergibt ein Reifen im Gesamtprodukt „Kraftfahrzeug“ bzw. in der Baugruppe „Rad“ nur dann Sinn, wenn auch eine Felge da ist, auf die der Reifen befestigt werden kann. Der Reifen kann also im Gesamtzusammenhang nie isoliert von der zugehörigen Felge betrachtet werden und ist somit dem Felgen-Bauteil untergeordnet. Die Radmuttern haben hier eine besondere Bedeutung, da sie mehrere Baugruppen miteinander verbinden („Rad“ und „Radaufhängung“). Welcher Baugruppe sie zuzurechnen sind, oder ob sie für sich alleine „außerhalb“ von Baugruppen modelliert werden, ist abhängig von der Designerin/dem Designer der Hierarchie. Hier ist erkennbar, dass es nicht unbedingt eine eindeutige logische Hierarchie zu einem mehrteiligen Produkt geben

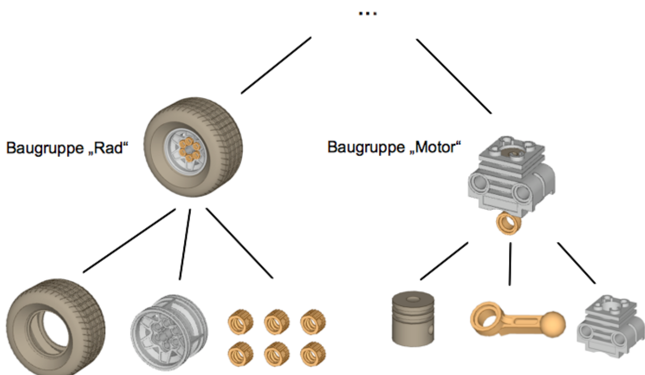


Abbildung 3.3: Produktbeschreibung anhand von funktionalen Baugruppen

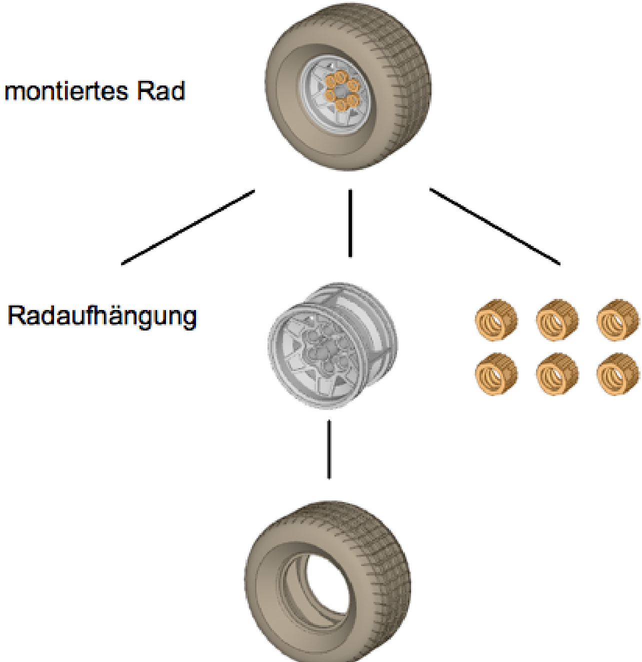


Abbildung 3.4: Logische Bauteil-Hierarchie

muss, sondern die Modelliererin/der Modellierer maßgeblichen Einfluss darauf hat. Eine logische Hierarchie kann sich auch am späteren Zusammenbauvorgang des Produktes orientieren.

- *Geometrische Bauteil-Hierarchie* Abhängigkeiten können sich auch aus geometrischen Zusammenhängen ergeben. Beispielsweise ist eine Radmutter in einer bestimmten Bohrung einer bestimmten Felge zentriert, fluchtet mit einem bestimmten Gewindestift, etc. Diese Art der Hierarchie findet sich oft in parametrischen CAD-

> Programmen. An einem Bauteil, der in einem virtuellen Koordinatensystem fixiert ist, wird ein weiterer Bauteil geometrisch ausgerichtet (z.B. über Flächenparallelität, Flucht von Bohrungen, Abstände) bis die Freiheitsgrade des hinzugekommenen Bauteils soweit eingeschränkt sind, dass dieser als geometrisch im Raum bestimmt gilt. Die dadurch entstehende geometrische Hierarchie muss dabei nicht notwendigerweise einer funktionalen oder logischen entsprechen, da es vom CAD-Zeichner abhängig ist, in welcher Reihenfolge Bauteile zusammengefügt werden.

Abbildung 3.5 zeigt einen Ausschnitt der Ontologie, der eine logische Hierarchie modellierbar macht. Individuen der Subklassen von `Description` (dt.: Beschreibung) dienen zur Beschreibung von Bauteilen. Im weiteren werden die einzelnen Klassen und deren Beziehungen zueinander näher beschrieben.

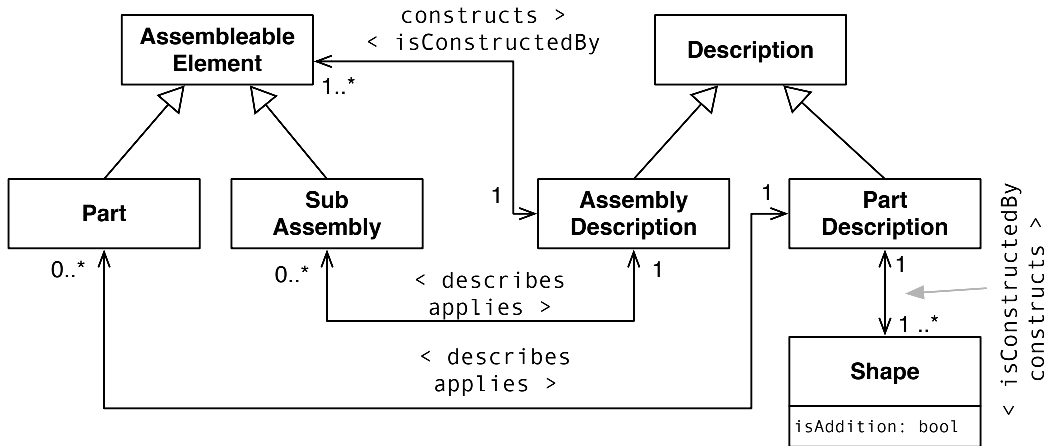


Abbildung 3.5: Ausschnitt aus der Ontologie: `AssembleableElement` (dt.: verbaubares Element) und `Description`

PartDescription

Ein Individuum der Klasse `PartDescription` (dt.: Bauteilbeschreibung) beschreibt einen spezifischen, atomaren - d.h. im Modell als unteilbar angesehenen - Bauteil. Es existiert in der Ontologie genau eine `PartDescription` pro Bauteil, egal wie oft dieser später im Produkt (oder mehreren Produkten) verwendet wird. Eine `PartDescription` bezieht sich dabei auf ein oder mehrere Individuen der Klasse `Shape` (dt.: Form), die über eine einfache Implementierung der „konstruktiven Festkörpergeometrie“ eine geometrische Definition eines Bauteils erlauben. Eine detailliertere Auseinandersetzung mit den verwendeten Geometrieelementen findet in Abschnitt 3.5 statt. Im Beispiel des Rades wäre für jeden verschiedenen Bauteil in der Teileliste (Abbildung 3.2) genau ein `PartDescription`-Individuum notwendig: eines für den Reifen, eines für die Radmutter und eines für die Felge.

AssemblyDescription

Ein Individuum der Klasse `AssemblyDescription` (dt.: Baugruppenbeschreibung) geht über die Beschreibung eines einzelnen Bauteils hinaus und beschreibt einen Teilverbund bzw. eine Baugruppe eines Produktes, im Beispiel etwa das komplette Rad. Wie im UML-Diagramm zu sehen ist, bezieht sich eine `AssemblyDescription` auf mehrere `AssembleableElement`s und nicht direkt auf die Beschreibungsklassen für Bauteile, d.h. die Beschreibung des Rades bezieht sich nicht direkt auf die Beschreibung von Felge, Reifen und Radmutter, sondern es sind weitere Individuen dazwischengeschaltet

Alle Teile und Baugruppen innerhalb einer `AssemblyDescription` gelten als gleichwertig, d.h. es gibt keine Reihenfolge. Muss eine Reihenfolge beim Zusammenbau eingehalten werden - etwa weil der Einbau eines Teils den Einbau eines anderen verhindern würde, dann muss diese über aufeinander aufbauende `AssemblyDescription`s im logischen Modell berücksichtigt werden.

Part und SubAssembly

Individuen der Subklassen von `AssembleableElement` stellen konkrete Manifestierungen einer Bauteilbeschreibung (Klasse `Part` (dt.: Bauteil)) oder eines Teilverbundes dar (Klasse `SubAssembly`). Für jede Anwendung (Instanz) einer Bauteilbeschreibung muss in der Ontologie jeweils ein Individuum der Klasse `Part` erzeugt werden, das sich auf eine ganz konkrete Teil-Beschreibung bezieht. Analog muss für jede Anwendung eines Teilverbundes ein `SubAssembly`-Individuum vorhanden sein. Bezogen auf das Rad-Beispiel werden für ein komplettes Rad ein Reifen, eine Felge und sechs Radmuttern benötigt. Daher müssen insgesamt acht Individuen vom Typ `Part` in der Ontologie erzeugt werden. In der `AssemblyDescription` des Rades kann nun auf diese acht `Part`-Individuen referenziert werden und so bilden diese gemeinsam einen Teilverbund.

Somit entspricht die `PartDescription` einem Konzept oder Bauplan für einen Bauteil und die `Part`-Individuen sind Software-Repräsentanten der nach diesem Bauplan entstandenen physisch vorhandenen Teile, die tatsächlich verbaut werden.

Soll nun ein Fahrzeug-Produkt mit Rädern ausgestattet werden, so ist es nicht notwendig in der `AssemblyDescription` für das Fahrzeug alle einzelnen Reifen, Felgen und Radmuttern zu referenzieren, sondern es genügt sich vier mal (bei einem Vierrad-Fahrzeug) auf den Rad-Teilverbund zu beziehen, d.h. auf vier `SubAssembly`-Individuen. Diese Vorgehensweise erlaubt es die Anzahl der benötigten Individuen in der Ontologie deutlich zu reduzieren.

3.1.1 Gesamtbeispiel

Eine Radfelge eines PKWs ist in einer `PartDescription` beschrieben. Diese kann Informationen zur Geometrie und weitere für die Felge wichtige Daten enthalten. Üblicherweise sind in einem PKW Felgen des selben Typs verbaut, daher genügt eine `PartDescription` um die grundlegenden Eigenschaften aller Felgen zu beschreiben. Um das Rad zu komplettieren, wird noch ein Reifen benötigt. Auch dieser ist über eine `PartDescription` in der

Ontologie zu definieren. Um nun den Verbund Felge-Reifen zu beschreiben, wird von jedem beteiligten Teil ein `Part`-Individuum benötigt und in einer `AssemblyDescription` vereint. Diese speichert eine Referenz auf alle Bauteile, die am Verbund beteiligt sind. Wird die Ontologie aus dem Blickwinkel eines relationalen Datenbankmodells betrachtet, würde dies einer many-to-many Beziehung zwischen einer `AssemblyDescription`-Relation und einer `PartDescription`-Relation entsprechen, wobei `Part` die Rolle der Surrogattabelle einnimmt. Eine `AssemblyDescription` kann sich nicht nur auf Einzelbauteile beziehen, sondern auch auf Baugruppen. Dazu gibt es analog zur Klasse `Part` in der Ontologie die Klasse `SubAssembly`. Sie erlaubt die Verwendung einer `AssemblyDescription` in einer weiteren `AssemblyDescription`. Soll nun das montierte Rad modelliert werden, sind mehrere Individuen notwendig:

- `AssemblyDescription` des Rades
- ein `SubAssembly` des Rades
- `PartDescription` einer Radmutter
- mehrere `Parts` für die tatsächlich verwendeten Radmuttern (hier vier Stück)
- `AssemblyDescription` des restlichen PKW
- ein `SubAssembly` des restlichen PKW

Daraus ergibt sich dann eine `AssemblyDescription` eines PKW mit *einem* montierten Rad. Für das nächste Rad kann nun diese `AssemblyDescription` verwendet und mit einem weiteren Rad-SubAssembly und sechs Radmuttern-`Parts` zu einer `AssemblyDescription` eines PKW mit *zwei* Rädern modelliert werden. Das dritte und vierte Rad werden dann analog angebracht (Abbildung 3.6). Dadurch ist die Reihenfolge, in der die Räder an das Fahrzeug angebaut werden schon durch die logische Hierarchie fest modelliert.

Es ist alternativ auch möglich die Montage aller Räder in einem Zusammenbau-Schritt zu modellieren, entsprechend werden dann vier `SubAssembly`s der Rad-AssemblyDescription benötigt, 24 Radmutter-`Parts`, und ein PKW-SubAssembly. Insgesamt sind damit auch weniger Individuen in der Ontologie, da es keine `AssemblyDescriptions` und SubAssemblies der Zwischenschritte gibt (Abbildung 3.7).

Allerdings sind diese beiden Modellvarianten nicht äquivalent. Dies liegt daran, dass in einer `AssemblyDescription` zwar Referenzen auf die verwendeten Teile in einem Attribut hinterlegt sind, in einer Ontologie ein Attribut mit mehreren Elementen aber keine Liste darstellt, sondern eine Menge - daher ist keine Reihenfolge definiert. Im Modell, bei dem alle Räder zusammen montiert werden, kann also nur beschrieben werden, welches Rad an welche Position kommt (diese Information ist im `Part` bzw. `SubAssembly` hinterlegt), aber nicht in welcher Reihenfolge die Räder an den PKW montiert werden müssen. Bei der zeitlichen Planung wird die Montage der Räder dadurch als parallelisierbar angesehen.

Die vorgestellte Ontologie erlaubt also verschiedene Modelle für das selbe Produkt, je nachdem welche Informationen für eine weitere Verarbeitung benötigt werden.

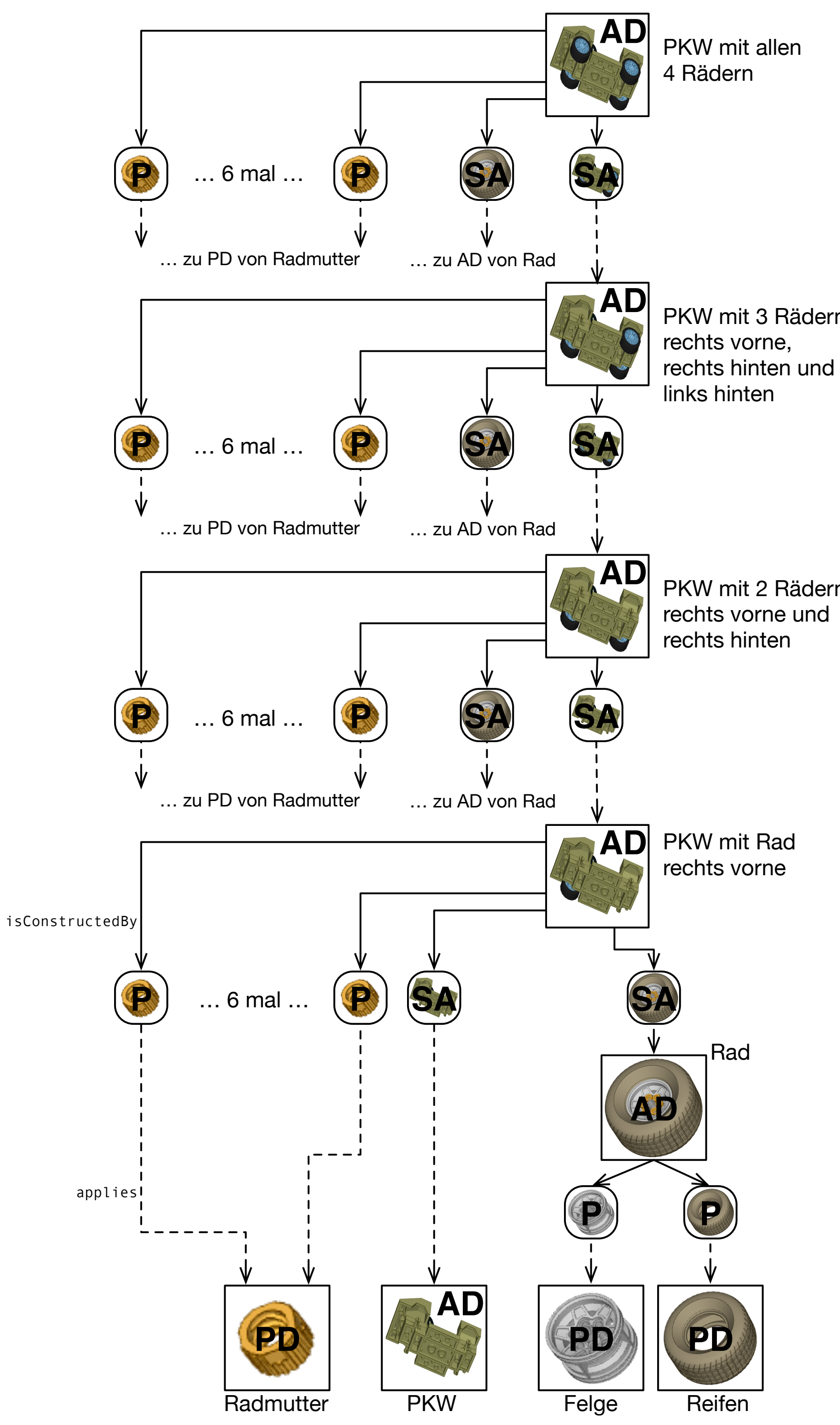


Abbildung 3.6: PKW mit vier Rädern: Zusammenhang zwischen `PartDescription` (PD), `Part` (P), `AssemblyDescription` (AD) und `SubAssembly` (SA)

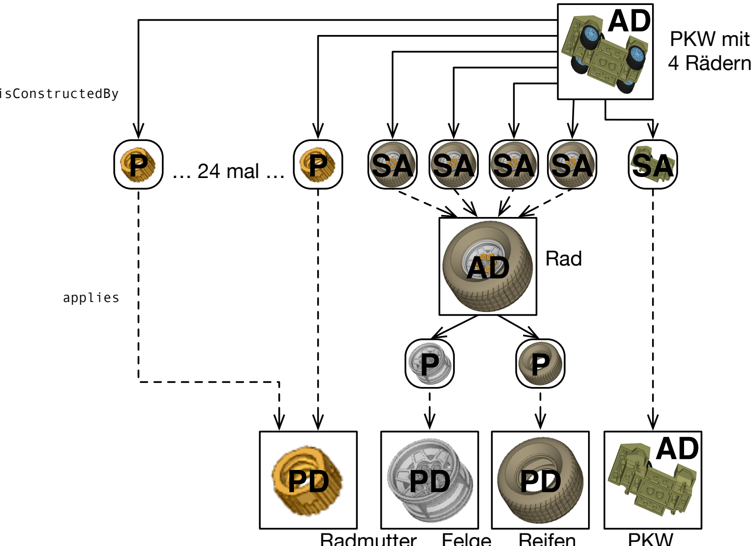


Abbildung 3.7: Alternative Version des PKW mit vier Rädern: alle Räder werden in einem Schritt montiert

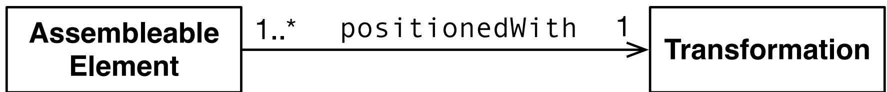


Abbildung 3.8: Jedes `AssembleableElement` ist über eine Transformationsmatrix geometrisch in einer Baugruppe verortet

3.1.2 Lage der Bauteile und Baugruppen

Jedes `AssembleableElement` kann innerhalb einer Baugruppe geometrisch verortet werden. Dies geschieht über eine Transformationsmatrix (Instanz der Klasse `Transformation` (dt.: Umwandlung), Abbildung 3.8). In einer solchen Transformationsmatrix sind Rotation und Verschiebung des Bauteils oder des Bauteilgruppe im gemeinsamen Koordinatensystem aller Bauteile spezifiziert (siehe auch Abschnitt 3.5).

Benutzerinnen und Benutzer der Ontologie sind selbst für eine sinnvolle Wahl des Koordinatensystems, des Ursprungs und der Achsenrichtungen verantwortlich und auch dafür, dies über alle Bauteile und Geometrieelemente konsistent zu halten. Was im ersten Moment wie eine zusätzliche Schwierigkeit im Umgang mit der Ontologie aussieht, kann genauer betrachtet oft dadurch gelöst werden, indem CAD-Daten zum Befüllen der Ontologie herangezogen werden, die ein Koordinatensystem in sich tragen. In Abschnitt 4.3 wird dies anhand von CAD-Daten für Lego-Modelle demonstriert.

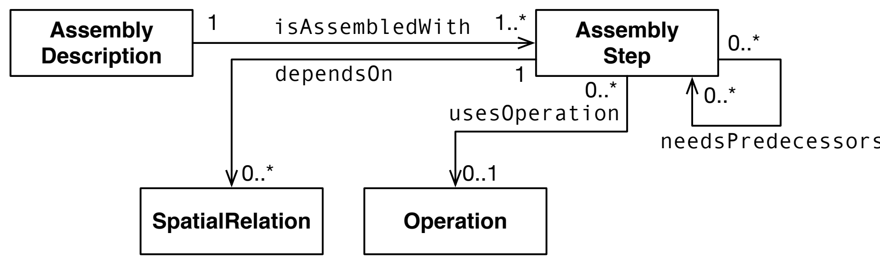


Abbildung 3.9: `AssemblyStep` bilden die Information für Zusammenbauschritte ab

3.2 Modellierung der notwendigen Schritte für den Zusammenbau

In den vorangegangenen Abschnitten wurde die Zusammengehörigkeit von Bauteilen und Bauteilgruppen beleuchtet. Dieser Abschnitt stellt Konzepte vor, die durchzuführenden Schritte und Aktionen beim Zusammenfügen von einzelnen Bauteilen zu modellieren.

Für diese Informationen können einer `AssemblyDescription` ein oder mehrere Individuen der Klasse `AssemblyStep` (dt.: Zusammenbauschritt) zugeordnet werden (Abbildung 3.9).

Ein `AssemblyStep` dient als Anknüpfungspunkt für Informationen zu den einzelnen Schritten und Operationen, die für die Herstellung eines Teileverbundes notwendig sind. Dazu verweist ein `AssemblyStep` auf ein Individuum der Klasse `Operation` (dt.: Arbeitsvorgang), das wiederum eine Fügungsoperation innerhalb der Ontologie darstellt. Die exakte Beschreibung verschiedener Fügungsoperationen ist nicht im Fokus dieser Arbeit, daher ist die Klasse `Operation` nicht näher spezifiziert. Sie dient als Schnittstelle für andere Ontologien, die diese näher beschreiben können. Beispielsweise beschreibt [32] eine Ontologie als Wissensbasis für eine flexible Fertigungsanlage, in der (unter anderem) auf den Fertigungsmaschinen durchführbare Fertigungsoperationen abgelegt sind (Abbildung 3.10). Über die Referenzierung eines `AssemblyStep`s mit einer `Operation` kann nun eine Verknüpfung zwischen beiden Ontologien hergestellt werden, sodass zu einem Zusammenbauschritt direkt die auf der Fertigungsanlage notwendigen Maßnahmen bekannt sind.

Beispiele für Zusammenbauoperationen wären etwa das zu verwendende Fügungsverfahren wie Schweißen, Kleben, etc. Bei solchen Verfahren sind Parameter notwendig, die den Fügungsprozess näher beschreiben. Beispiele sind hierfür Fügungstemperaturen oder benötigtes Zusatzmaterial wie Klebstoffe. Hier muss auch entschieden werden, was sind Zusatzmaterialien - also keine Bauteile im Sinne der Assembling-Ontologie - und daher nur als Attribut eines Fügungsprozesses beschrieben und wo müssen Zusatzmaterialien eventuell als eigenständige Bauteile modelliert werden. Bei einer Verschraubung wird es für manche Anwendungsfälle genügen, die Schraube, benötigte Unterlegscheiben, Muttern und Schraubensicherungen als Attribute des Fügungsprozesses zu modellieren. Dient das Assembling-Modell beispielsweise dazu, um im Recyclingprozess zu beschreiben, wie

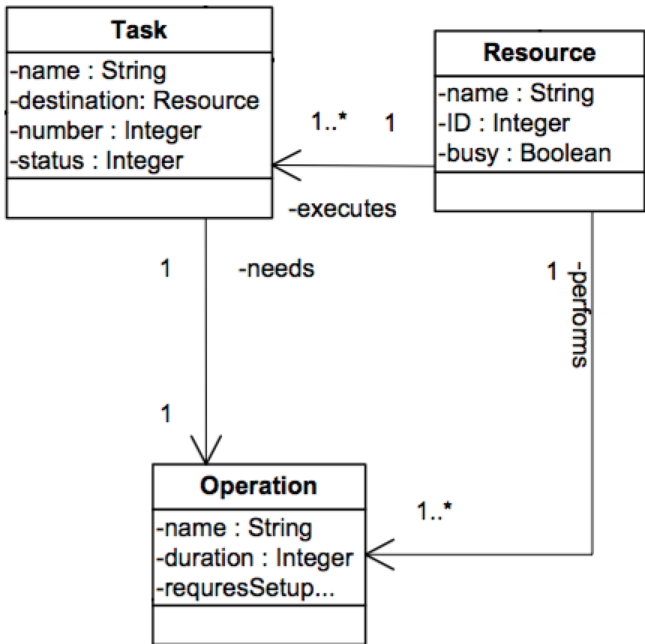


Abbildung 3.10: Fertigungsoperationen in einer Ontologie für flexibel Fertigungsanlagen (Bildausschnitt [32])

ein Bauteil händisch zerlegt werden muss und welche Werkzeuge dazu benötigt werden, genügt eine simple Attributinfomation wie „Schraube M6, 30mm lang, Sechskantkopf“ oder „Schraube ISO 4014, M6x30“. Die geometrischen Informationen, wie diese Schraube entfernt werden muss, ergeben sich für das zerlegende Recycling-Personal dann am Bauteil. Für eine automatisierte Zerlegung, bei der eine Maschine die Aufgabe des Trennens der Bauteile übernehmen soll, werden geometrische Angaben über Lage, Ausrichtung, Anzugsmoment etc. auch benötigt. Hier ist es von Vorteil, die Schraube wie ein eigenes Bauteil in der Ontologie zu modellieren, da dann Angaben zur Lage und Bauteilgeometrie möglich sind.

`AssemblySteps` dürfen auch voneinander abhängig sein. Dazu können Instanzen über die Eigenschaft `needsPredecessor` (dt.: braucht Vorgänger) an andere Bearbeitungsschritte gekoppelt werden (Abbildung 3.9). So lässt sich eine Fügung, die aus mehreren getrennten Schritten besteht, in der Ontologie abbilden.

Ein Beispiel soll die Rolle der `AssemblySteps` verdeutlichen. Abbildung 3.11 zeigt eine Variante der Beschreibung eines Rades. In diesem Beispiel umfasst zwecks Einfachheit die `AssemblyDescription` nur Felge und Reifen. Diese referenziert auf die drei Zusammenbauschritte „Felgenränder einfetten“, „Reifen aufziehen“ und „wuchten“, die den Zusammenbau näher beschreiben. Da diese Schritte nicht in beliebiger Reihenfolge ausgeführt werden können, ist über die Eigenschaft `needsPredecessor` bei jedem Schritt sein unmittelbarer Vorgänger angegeben. Dadurch ergibt sich eine Schrittkette, in der die

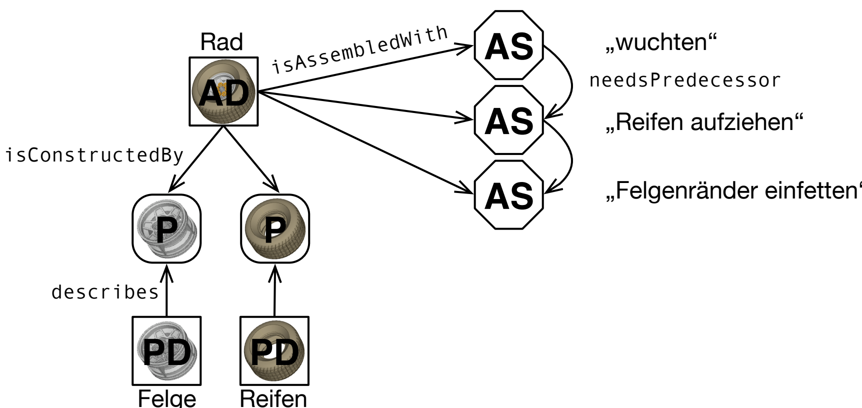


Abbildung 3.11: Beispiel einer `AssemblyDescription` (AD) mit zugeordneten `AssemblySteps` (AS)

Verarbeitung ablaufen muss.

Die exemplarischen Bearbeitungsschritte in Abbildung 3.11 bilden eine strenge Reihenfolge, d.h. jeder Schritt außer dem ersten hat genau einen Vorgänger. Das muss aber nicht für jeden Zusammenbau gelten. In Abbildung 3.12 sind drei andere Fälle aufgezeigt. In (a) benötigt der Arbeitsschritt für die Befestigung des Stegs zwei Vorgängerschritte, welche die Befestigungsstifte in den Steg einpassen. Ein Arbeitsschritt hat hier mehrere Vorgänger. Da es für diese Vorgängerschritte keine Aussage über eine gegenseitige Reihenfolge gibt, werden diese als parallelisierbar angesehen. Dies spielt später bei der Planung des Produktes eine Rolle. In (b) gibt es mehrere Arbeitsschritte (Schrauben befestigen), die einen gemeinsamen Vorgänger haben. Die einzelnen Verschraubungsschritte werden wiederum als parallelisierbar angesehen. In (c) gibt es überhaupt keine Abhängigkeiten zwischen den einzelnen Arbeitsschritten. Sämtliche dieser Schritte können bei der Produktion gleichzeitig oder in beliebiger Reihenfolge durchgeführt werden. Alle in Abbildung 3.12 gezeigten Fälle dürfen für eine `AssemblyDescription` kombiniert werden. So können auch sehr komplexe Bearbeitungen modelliert werden.

`AssemblySteps` gelten aber immer nur für genau eine `AssemblyDescription`. Daher ist es z.B. bei der gezeigten Modellierung nicht vorgesehen, die Räder aus (c) schon während der Montage der Platte in (b) zu montieren.

3.3 Produkte und die Planung des Zusammenbaus

Mit Hilfe der in der Ontologie abgelegten Daten ist es möglich, eine sequenzielle zeitliche Abfolge der notwendigen Schritte abzuleiten, um den Zusammenbau beispielsweise für eine flexible Fertigungsanlage planen zu können. Dazu wurden mehrere Ableitungsregeln definiert. Diese Regeln beziehen sich direkt auf Fakten, die in der Ontologie abgelegt sind und erzeugen daraus die Planungsschritte. Diese müssen daher von Benutzerinnen und

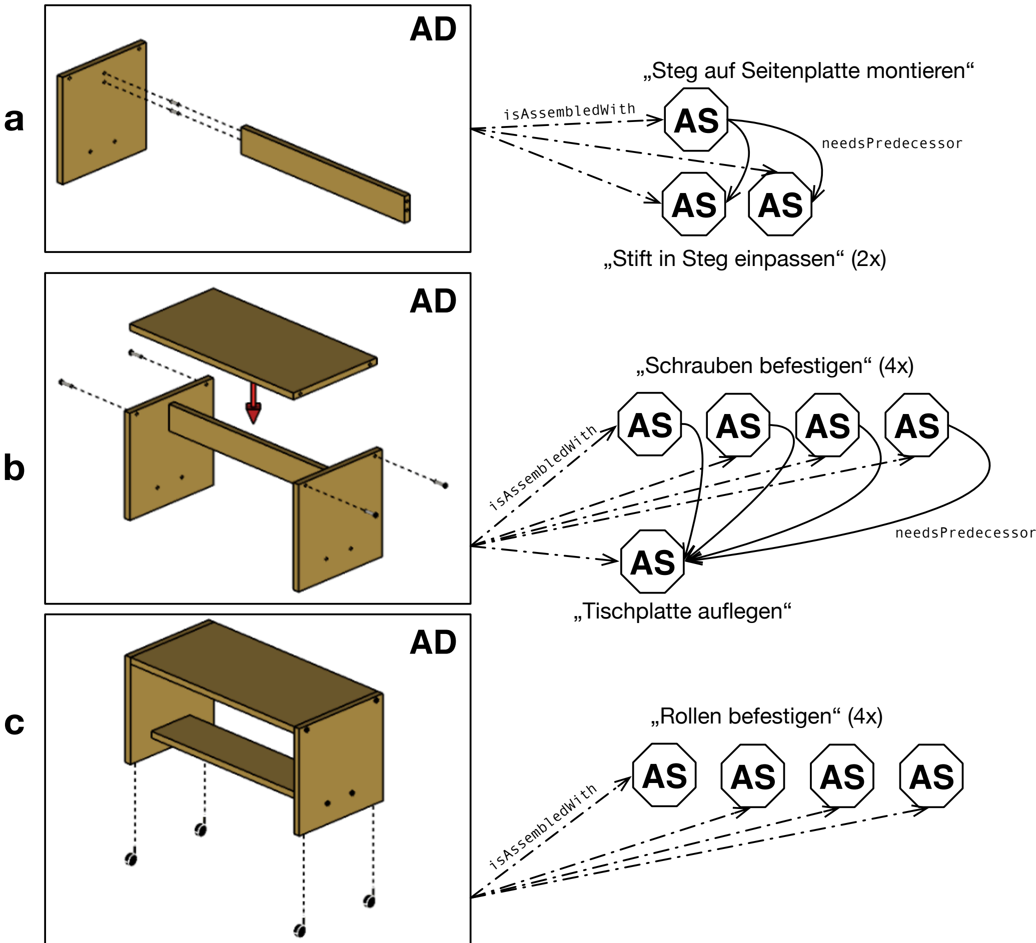


Abbildung 3.12: Varianten für die Verkettung von `AssemblySteps` (Grafiken des Tisches aus [30])

Benutzern der Ontologie nicht manuell angelegt werden. Ableitungsregeln sind in der Form „Wenn-Dann“ notiert. Im „Wenn“-Teil entscheidet eine Bedingung darüber, auf welche Fakten diese Regel angewendet wird. Werden passende Daten gefunden werden für diese alle Aktionen aus dem „Dann“-Teil ausgeführt - dies wird auch als „feuern der Regel“ bezeichnet.

Abbildung 3.13 zeigt alle an der Planung beteiligten Klassen innerhalb der Ontologie. Von zentraler Bedeutung ist die Klasse `SchedulingStep`, die jeden `AssemblyStep` innerhalb des Zusammenbauvorgangs zeitlich einordnet. Dazu bekommt jeder `SchedulingStep` eine eindeutige Nummer zugewiesen. Werden die Planungsschritte in numerisch aufsteigender Reihenfolge abgearbeitet, so ergibt sich ein fertiges Produkt.

Zum Anstoßen einer Planung des Zusammenbaus muss zuerst eine Instanz der Klasse `Product` angelegt werden. Diese verweist auf ein `SubAssembly`, das die Baugruppe be-

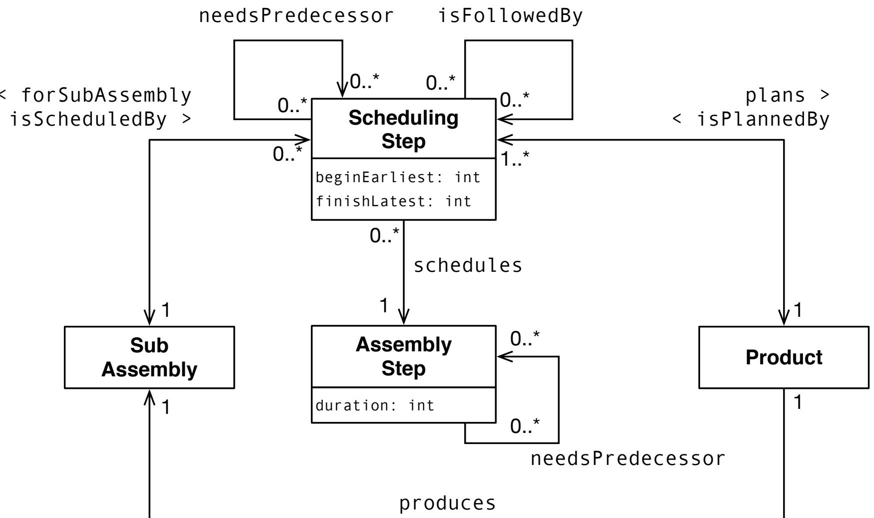


Abbildung 3.13: Ausschnitt aus der Ontologie mit den Klassen zur Planung des Zusammenbaus

schreibt welche produziert werden soll. Diese Beschreibung kommt von einer referenzierten Instanz von `AssemblyDescription`. Danach kann das zugehörige Regelwerk aktiviert werden (siehe auch Abschnitt 4.6), welches einzelne, miteinander über die Beziehungen `needsPredecessor` und `isFollowedBy` (dt.: gefolgt von) verknüpfte `SchedulingStep`-Instanzen erzeugt.

Zusammenhang zwischen AssemblySteps und SchedulingSteps

Wie in Abschnitt 3.2 beschrieben, beziehen sich `AssemblySteps` auf genau eine Bauteilgruppe. Sie modellieren die notwendigen Maßnahmen um die beschriebene Bauteilgruppe zu fügen. Abbildung 3.6 erweitert das Beispiel aus Abbildung 3.11 auf ein Fahrzeug mit vier Rädern. Für dieses werden vier `SubAssemblys` für die `AssemblyDescription` „Rad“ angelegt. Bis zum fertigen Fahrzeug muss also vier mal „Felgenränder eingefettet“, „Reifen aufziehen“ und „wuchten“ durchgeführt werden.

`SchedulingSteps` verbinden nun diese Arbeitsschritte in einer Folge. Abbildung 3.14 zeigt einen Ausschnitt aus Abbildung 3.6 ergänzt um die zugehörigen Arbeitsschritte. Mit Hilfe der Regeln werden nun Planungsschritte (`SchedulingStep`) erzeugt. Ein mögliches Ergebnis ist in Abbildung 3.14 rechts zu sehen. Die erzeugten Nummern der `SchedulingSteps` geben eine mögliche Folge der Schritte von der kleinsten zur größten Nummer vor. In jedem `SchedulingStep` sind die unmittelbaren Vorgänger- und Nachfolgerschritte über die Beziehungen `needsPredecessor` und `isFollowedBy` referenziert. Im gezeigten Fall haben sich die Regeln dafür entschieden, zuerst die Arbeitsschritte der (sehr stark vereinfachten) PKW-Baugruppe zu planen und erst danach die des Rades. Dies wurde willkürlich so festgelegt, da keine Informationen in der Ontologie hinterlegt sind, welche Baugruppe

```
1 Wenn
2   - ein AssemblyStep ohne Nachfolger in der Faktenbasis existert
3 Dann
4   - erzeuge einen SchedulingStep und verknüpfe diesen mit dem
      AssemblyStep und dem aktuell bearbeiteten Produkt.
5   - gib dem SchedulingStep eine fortlaufende Nummer
6   - lösche den gerade bearbeiteten AssemblyStep aus der
      Faktenbasis - er wird nicht mehr benötigt.
```

Listing 1: Erstellung der `SchedulingSteps`

hier zuerst behandelt werden soll. In jedem Fall wird eine funktionierende Reihenfolge erzeugt. Auf eine mögliche Parallelisierung wird in Abschnitt 3.4 eingegangen.

Die Vorgehensweise beim Ermitteln der Reihenfolge wird im Folgenden beschrieben. Zuerst werden alle `AssemblySteps`, die für die Erzeugung eines Produktes notwendig sind in die Faktenbasis des Regelsystems geladen. Eine Regel feuert auf jeden `AssemblyStep`, der selber keine Nachfolger hat - also in einer Kette von Bearbeitungsschritten innerhalb eines `SubAssemblys` als letzter ausgeführt wird. Im in Abbildung 3.14 gezeigten Beispiel sind das die Bearbeitungsschritte „schleifen“ von der Baugruppe „PKW“ und „wuchten“ von der Baugruppe „Rad“. Für diese `AssemblySteps` werden nun `SchedulingSteps` erzeugt und verknüpft. Eine laufende Nummer wird dabei zwischen den Regelausführungen weitergegeben. Sie ist später die Basis für die vorgeschlagene Erzeugungsreihenfolge des Produktes. Welcher der beiden zuerst an die Reihe kommt ist nicht festgelegt und kann über mehrere Durchläufe der Regeln auch variieren. Zuletzt werden die Bearbeitungsschritte aus der Faktenbasis entfernt. Dadurch werden jetzt die Schritte „Rahmen schweißen“ und „Reifen aufziehen“ zu Bearbeitungsschritten ohne Nachfolger und die Regel kann erneut feuern. Dies setzt sich solange fort, bis alle Arbeitsschritte abgearbeitet wurden. Listing 1 zeigt die Vorgehensweise in Pseudocode.

Damit wird für alle notwendigen `AssemblySteps` jeweils ein `SchedulingStep` erzeugt. Als nächsten Schritt müssen die `SchedulingSteps` noch miteinander in Beziehung gesetzt werden. Dabei sind zwei Fälle zu unterscheiden:

- *Zu verknüpfende `SchedulingSteps` beziehen sich auf `AssemblySteps`, die alle der selben Baugruppe zugeordnet sind.* D.h. alle diese `AssemblySteps` sind notwendig um *eine* Fügung durchzuführen.

- *Zu verknüpfende `SchedulingSteps` beziehen sich auf unterschiedliche Baugruppen.* Sie sind also die Bindeglieder zwischen den Zusammenbauschritten verschiedener Fügungen. Dieser Fall ist deutlich komplexer zu handhaben.

Im ersten Fall genügt es, die Beziehungen der `AssemblySteps` der gemeinsamen Baugruppe auf die `SchedulingSteps` zu übertragen (Listing 2). Eine Erweiterung ergibt sich dadurch,

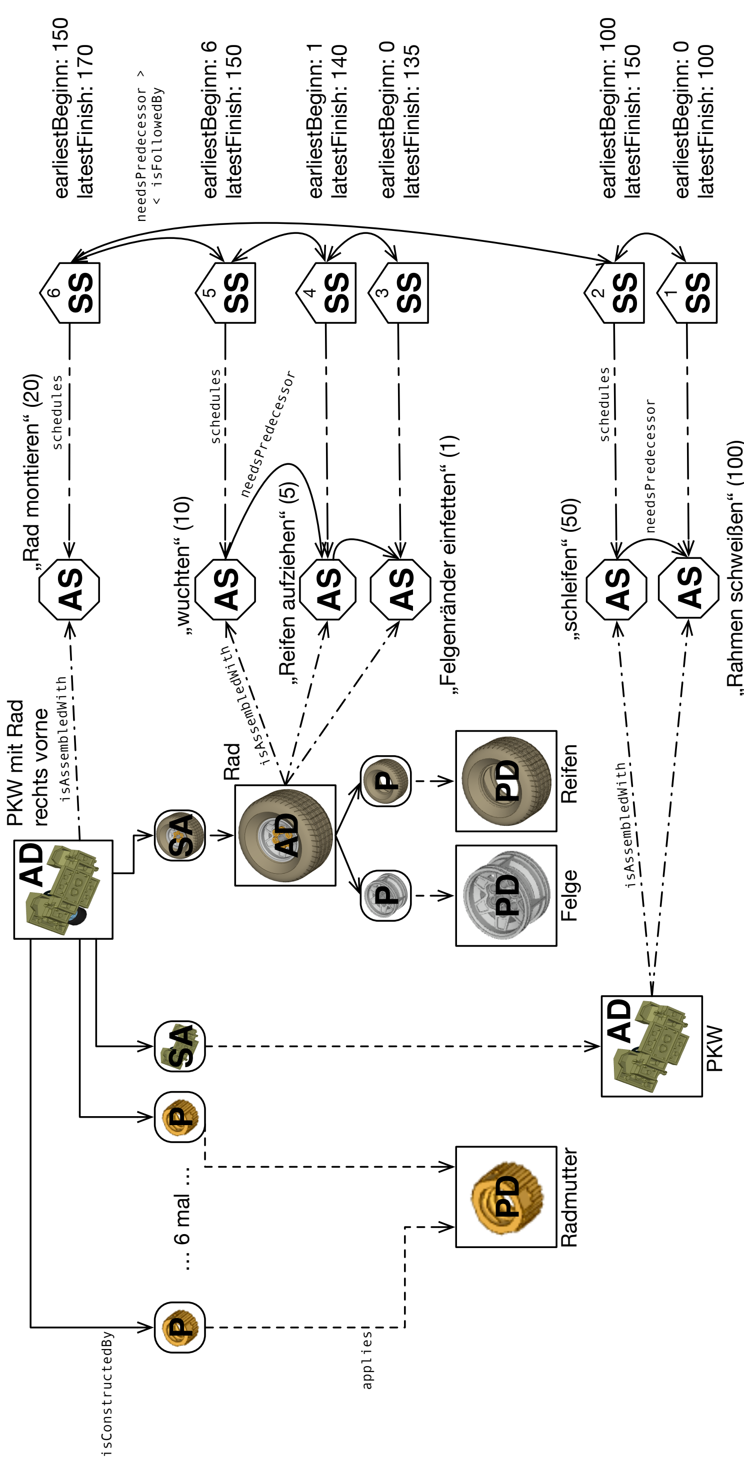


Abbildung 3.14: Montage des ersten Rades des Fahrzeuges mit zugehörigen `AssemblySteps` (AS) und daraus generierten `SchedulingSteps` (SS).

```
1 Wenn
2   – ein 1. SchedulingStep in der Faktenbasis existert
3   – und ein 2. SchedulingStep, dessen verknüpfter AssemblyStep
      den AssemblyStep des 1. Schedulingsteps als Vorgänger hat,
      aber selber nicht schon Vorgänger des 1. SchedulingSteps ist
4   – beide SchedulingSteps sich auf das selbe SubAssembly beziehen
5 Dann
6   – setze den 1. SchedulingStep als Vorgänger des 2. und den 2.
      als Nachfolger des 1. Die Beziehung der AssemblySteps
      untereinander wird also auf die Schedulingsteps übertragen.
```

Listing 2: Beziehungen der `SchedulingSteps` für eine Baugruppe ermitteln

dass bei den `SchedulingSteps` die Beziehungen in beide Richtungen angegeben werden, also nicht nur auf Vorgänger verwiesen wird, sondern auch auf Nachfolger.

Der zweite Fall ist komplexer. Grundsätzlich lassen sich Verknüpfungen der Planungsschritte zwischen verschiedenen Baugruppen über eine Regel wie sie Listing 3 zeigt herstellen. Ein besondere Situation ergibt sich mit `AssemblyDescriptions`, die sich aus mehreren *gleichen* Baugruppen zusammensetzen. Hier finden die Regeln in der Faktenbasis mehrmals `SchedulingSteps` vor, die sich auf die gleichen `AssemblySteps` beziehen. Es muss sichergestellt werden, dass alle diese `SchedulingSteps` auch wirklich in die Bauteilsequenz integriert werden, und nicht nur einer als „Repräsentant" für alle „gleichen" Planungschritte ausgewählt wird. Sonst würden im Ergebnis ganze Baugruppen fehlen. In der Regeln in Listing 3 wird dies über die letzte Bedingung im Wenn-Teil realisiert (Zeile 5).

3.4 Parallelisierung

In der Realität werden PKW und Rad mit hoher Wahrscheinlichkeit unabhängig voneinander gefertigt werden, d.h. eine Fertigungsanlage wird den PKW-Rahmen schweißen und schleifen und eine andere (vielleicht sogar in einer anderen Firma) die Räder vorfertigen. Es gibt keine Abhängigkeiten zwischen den notwendigen Schritten um den PKW einerseits und das Rad andererseits zusammenzubauen. Wichtig ist nur, dass die fertigen Baugruppen zu bestimmten Zeitpunkten für eine Montage zur Verfügung stehen. Genau dafür berechnen die Regeln zu jedem Planungsschritt:

- *wann dieser frühestmöglich begonnen werden kann* und
- *wann dieser spätestens abgeschlossen sein muss.*

Bei den Arbeitsschritten kann im Attribut `duration` (dt.: Dauer) der Klasse `AssemblyStep` eine Dauer in ganzzahligen Zeiteinheiten (z.B. Sekunden oder Takte einer Fertigungsan-

```
1 Wenn
2   – ein 1. SchedulingStep einer Baugruppe (a) existiert, der
      selber kein Vorgänger bei einem anderen SchedulingStep der
      selben Baugruppe ist
3   – und gibt es einen 2. SchedulingStep einer anderen Baugruppe
      (b), der den 1. SchedulingStep noch nicht als Vorgänger hat
      (dies verhindert ein Endlosfeuern der Regeln) und auch keinen
      Vorgänger in der eigenen Baugruppe hat (d.h. dieser
      Schedulingstep bezieht sich auf einen AssemblyStep, der ein
      Erster in der Baugruppe ist)
4   – und baut Baugruppe (b) auf Baugruppe (a) auf
5   – und gibt es keinen 3. SchedulingStep, der den 1. schon als
      als Vorgänger verlinkt hat (damit wird erreicht, dass sich
      mehrere SchedulingSteps, die sich auf den selben AssemblyStep
      beziehen, gleichmäßig auf ihre Nachfolger aufteilen und
      nicht alle auf den selben referenzieren)
6 Dann
7   – setze den 1. SchedulingStep als Vorgänger des 2. und den 2.
      als Nachfolger des 1.
```

Listing 3: Beziehungen der `SchedulingSteps` zwischen verschiedenen Baugruppen ermitteln

lage) angegeben werden (siehe auch Abbildung 3.13). Werden keine Werte für die Dauer hinterlegt, so wird eine fiktive Dauer von 1 verwendet.

Im Attribut `finishLatest` (dt.: beende spätestens) des letzten `SchedulingStep` lässt sich auch die Dauer der Montage des gesamten Produktes ablesen.

Frühestmöglicher Beginn

Eine geplante Produktmontage beginnt immer zum Zeitpunkt 0. Der frühestmögliche Zeitpunkt für einen Zusammenbauschritt ist also eine relative Zeitangabe und leitet sich aus den Montagezeiten der Vorgängerschritte ab, unter der Annahme, dass diese immer zu ihrem jeweiligen frühestmöglichen Zeitpunkt begonnen und wo möglich parallel ausgeführt werden.

In Abbildung 3.14 sind exemplarische Zeiteinheiten neben den Arbeitsschritten notiert. Beispielsweise bedeutet „schleifen“ (50), dass für das Schleifen des PKW-Rahmens fünfzig Zeiteinheiten vorgesehen sind. Der Arbeitsschritt „schleifen“ benötigt als Vorgänger die Fertigstellung des Arbeitsschrittes „Rahmen schweißen“. Dieser beginnt bei Zeitpunkt 0 und dauert 100 Zeiteinheiten. Der frühestmögliche Zeitpunkt für den Beginn von „schleifen“ ist somit bei Zeitpunkt 100, wenn „Rahmen schweißen“ fertiggestellt ist. Werden die Arbeitsschritte des Rades betrachtet, so kann auch dort der erste Arbeitsschritt („Felge einfetten“) bei Zeitpunkt 0 beginnen, denn es gibt keine Vorgängerarbeitsschritte. Der darauf aufbauende Schritt kann bei Zeitpunkt 1 beginnen und der letzte („wuchten“)

```
1 Wenn
2   – ein SchedulingStep mit mindestens einem Vorgänger in der
      Faktenbasis existert
3   – der noch keinen frühestmöglichen Beginn > 0 gesetzt hat
4 Dann
5   – setze den frühestmöglichen Beginn auf den spätesten
      Zeitpunkt, zu dem alle Vorgänger des SchedulingSteps fertig
      sein müssen (dies ergibt sich aus dem frühestmöglichen Beginn
      und der Dauer der Vorgängerschritte)
6   – setze bei allen Nachfolgern den frühestmöglichen Beginn auf 0
      –> damit wird dieser für die Nachfolger neu berechnet
```

Listing 4: Berechnung des frühestmöglichen Beginns eines Planungsschrittes

letztlich zu Zeitpunkt 6. Der letzte Arbeitsschritt „Rad montieren" benötigt die Fertigstellung sowohl von Rad als auch vom PKW. Der frühestmögliche Zeitpunkt hängt also davon ab, wann diese fertig werden. Da das Rad bereits zum Zeitpunkt 16 fertig wird (Dauer aller Arbeitsschritte des Rades zusammen), der PKW-Rahmen aber erst zu Zeitpunkt 150, kann somit mit dem Rad montieren erst zu Zeitpunkt 150 begonnen werden.

Um den frühestmöglichen Beginn zu ermitteln wird nur eine Regel benötigt. Diese wird von allen `SchedulingSteps` ausgelöst, die Vorgänger haben. Da `SchedulingSteps` ohne Vorgänger immer den frühestmöglichen Beginn 0 haben, brauchen diese nicht extra verarbeitet werden - das Attribut ist standardmäßig bereits auf 0 gesetzt. Die Regel bestimmt aus allen Vorgängern, welcher am Längsten für seine Tätigkeit braucht. Diese Information wird dann als neuer Wert für das Attribut **`beginEarliest`** (dt.: beginne frühestens) verwendet (Listing 4).

Spätestmögliche Fertigstellung

Für eine flexible Fertigungsplanung ist es wichtig zu wissen, wie lange mit gewissen Fertigungsschritten gewartet werden kann. Dies ermöglicht beispielsweise, Tätigkeiten zu einem späteren Zeitpunkt auszuführen, an dem die benötigten Geräte nicht ausgelastet sind. Dieses Hinauszögern darf aber die Fertigstellung des geplanten Produktes nicht verzögern, daher wird für jeden Planungsschritt ermittelt, wann dieser spätestens abgeschlossen sein muss, um im Plan zu bleiben.

Aus den in Abbildung 3.14 dargestellten Zeiten für die Arbeitsschritte des PKW ergibt sich, dass es bei der Erstellung des PKW-Rahmens keinen Spielraum für Verzögerungen gibt. Der spätestmögliche Zeitpunkt von „Rahmen schweißen" ist genau nach Ablauf der benötigten Dauer (100). Genauso darf es bei „schleifen" keine Verzögerungen geben. Der spätestmögliche Zeitpunkt für die Fertigstellung des gesamten `SubAssemblys` PKW (150) ergibt sich daher aus der Summe der einzelnen Zeitspannen jedes Arbeitsschrittes.

Beim Rad ergibt sich eine andere Situation. Da für die Montage eines Rades sehr viel

```
1 Wenn
2   - ein SchedulingStep ohne Nachfolger in der Faktenbasis existert
3   - der schon einen frühestmöglichen Beginn > 0 gesetzt hat und
4   - dessen spätestmöglicher Fertigstellungszeitpunkt aber davor
      liegt
5 Dann
6   - setze den spätestmöglichen Fertigstellungszeitpunkt auf den
      frühesten Beginn + die definierte Dauer für den geplanten
      Schritt
7   - setze bei allen Vorgängern den spätestmöglichen
      Fertigstellungszeitpunkt auf 0 -> damit wird dieser für die
      Vorgänger neu berechnet
```

Listing 5: Start der Berechnung des spätestmöglichen Fertigstellungszeitpunktes

weniger Zeit benötigt wird als für den PKW, kann die Vormontage des Rades später beginnen. Es muss nur sichergestellt sein, dass zum Zeitpunkt des frühestmöglichen Beginns der Radmontage (Zeitpunkt 150), das Rad fertig zusammengebaut vorliegt. Somit kann beispielsweise „Reifen aufziehen“ irgendwann zwischen Zeitpunkt 1 und Zeitpunkt 135 begonnen werden, ohne dass die Fertigstellung des gesamten PKW mit Rad verzögert würde.

Eine Angabe des spätestmöglichen Beginns anstatt der spätestmöglichen Fertigstellung ist ebenso möglich. Da sich das eine aber leicht aus dem anderen berechnen lässt, wurde auf eine redundante Speicherung verzichtet. Die spätestmögliche Fertigstellung zu erfassen ist sinnvoller, da so die maximal mögliche Zeitspanne für einen Arbeitsschritt direkt ersichtlich ist.

Auch kann nicht für alle Arbeitsschritte deren ganzer zeitlicher Spielraum gleichzeitig genutzt werden. Ein verzögerter Start eines Schrittes führt natürlich dazu, dass Nachfolger einen späteren möglichen Beginnzeitpunkt haben. Durch Setzen des Attributes `beginEarliest` auf die tatsächliche Startzeit (diese muss allerdings größer sein, als der ursprüngliche Wert von `beginEarliest`) und ein erneutes Anstoßen der Regelverarbeitung, können alle nachfolgenden Schritte neu berechnet werden.

Um den spätestmöglichen Fertigstellungszeitpunkt zu ermitteln, werden mehrere Regeln benötigt. Die Berechnung wird mit den letzten `SchedulingStep` für ein Produkt begonnen. Bei diesen berechnet sich der korrekte Wert aus dem frühesten Beginnzeitpunkt plus der Dauer, die dieser Schritt benötigt (Listing 5).

Jetzt können die Planungsschritte bis zum Anfang des Produktionsablaufes zurückgegangen und Schritt für Schritt der spätestmögliche Zeitpunkt ergänzt werden. Dazu dient eine weitere Regel. Sie funktioniert ähnlich wie die Regel zum Setzen des frühestmöglichen Beginns. Aus allen Nachfolgern eines Planungsschrittes wird der spätestmögliche Start ermittelt. Der früheste Wert davon ist der neue spätestmögliche Fertigstellungszeitpunkt des betrachteten Planungsschrittes (Listing 6).

```
1 Wenn
2   – ein SchedulingStep mit mindestens einem Nachfolger in der
      Faktenbasis existert
3   – der schon einen frühestmöglichen Beginn > 0 gesetzt hat und
4   – dessen spätestmöglicher Fertigstellungszeitpunkt aber noch 0
      ist
5 Dann
6   – setze den spätestmöglichen Fertigstellungszeitpunkt auf den
      kleinsten frühesten Beginn aller Nachfolger des geplanten
      Schrittes
7   – setze bei allen Vorgängern den spätestmöglichen
      Fertigstellungszeitpunkt auf 0 –> damit wird dieser für die
      Vorgänger neu berechnet
```

Listing 6: Berechnung der spätestmöglichen Fertigstellungszeitpunkte für alle weiteren Planungsschritte

Als Resultat ist bei jedem Planungschritt ein Zeitbereich angegeben, in dem die Fertigung letztlich passieren muss. Es lässt sich sofort auslesen, welche Planungsschritte verzögert werden dürfen, ohne den Gesamtplan zu gefährden. Genauso sind auch die „kritischen Pfade“ eines Zusammenbaus ersichtlich, also Ketten jener Planungsschritte vom Beginn zum fertigen Produkt die keinen Spielraum haben, d.h. wo frühestmöglicher Beginn und spätestmöglicher Fertigstellungszeitpunkt genau um die Dauer aller zugehörigen Arbeitsschritte auseinanderliegen.

3.5 Geometriemodellierung

Für Fügungen sind Merkmale der beteiligten Teile interessant, die bei der Fügung miteinander in eine (geometrische) Beziehung gesetzt werden müssen. Ein Beispiel ist das parallele Ausrichten von zwei Flächen bei einer Klebung. In der Ontologie können solche Beziehungen über die Klasse `SpatialRelation` (dt.: räumliche Beziehung) und deren Subklassen definiert werden (Abbildung 3.15). Über die Beziehung `dependsOn` (dt.: hängt ab von) können diese Informationen mit einem Zusammenbauschritt verknüpft werden.

Eine `SpatialRelation` verknüpft mehrere Primitive wie Punkte (Klasse `Vertex`), Kanten (Klasse `Edge`) und Flächen (Klasse `Area`), wobei Flächen sich aus mindestens drei Kanten definieren und Kanten aus zwei Punkten. Subklassen von `SpatialRelation` können Einschränkungen definieren, die für beteiligte Primitive gelten müssen, z.B. zwei Kanten müssen parallel sein. Ein Reasoner - eine Software, die aus Fakten in der Ontologie durch logische Schlüsse neue Fakten generiert - kann alle in der Ontologie hinterlegten Primitive auf Zugehörigkeit zu einer dieser Klassen überprüfen.

Primitive ergeben sich aus der Bauteilgeometrie. Daher wurde eine Möglichkeit geschaffen, relevante Teile der Bauteilgeometrie ebenfalls in der Ontologie abzulegen und die Primitive mit Hilfe von Regeln daraus abzuleiten.

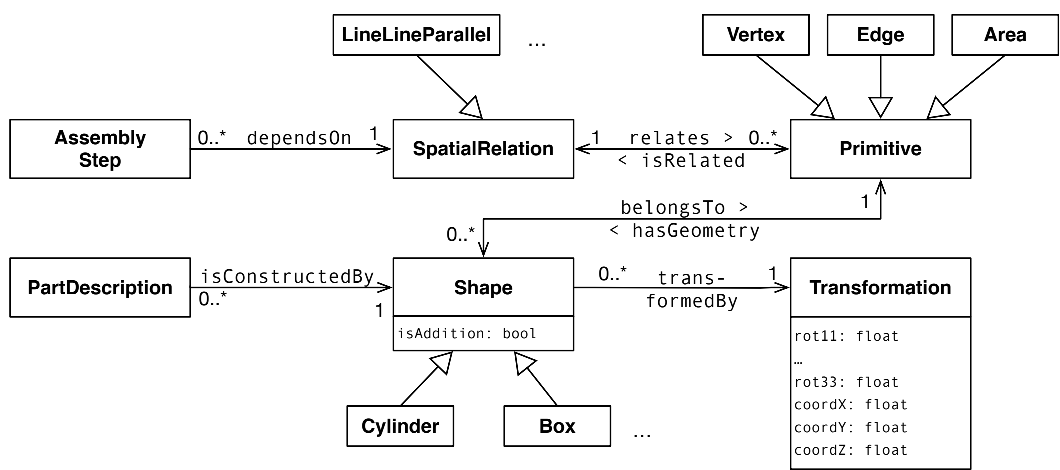


Abbildung 3.15: Geometrieklassen und deren Beziehungen

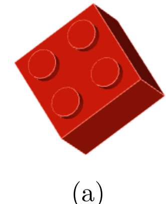
3.5.1 Shape

Für die Definition der Bauteil-Körper dienen Individuen der Klasse `Shape`. Das sind mathematisch und geometrisch einfach zu beschreibende Grundkörper wie Quader oder Zylinder, die in einer `PartDescription` mit Hilfe der Methode „Konstruktive Festkörpergeometrie“ (CSG - Constructive Solid Geometry) zu komplexeren Körpern verbunden werden können. Dabei werden alle in einer `PartDescription` referenzierten Geometrien als vereinigt angesehen. Über das Attribut `isAddition` (dt.: ist Erweiterung) kann auch eine Differenz festgelegt werden, d.h. eine `Shape`, für die das Attribut `isAddition` auf *false* gesetzt ist, wird von der restlichen Geometrie „abgezogen“.

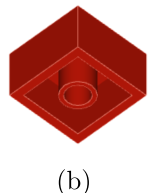
Ein klassischer „Vierer-Legostein“ (Abbildung 3.16a) kann aus einem Quader- und vier Zylinder-Shapes beschrieben werden. Soll der Hohlraum an der Unterseite auch modelliert werden (Abbildung 3.16b), wird die `PartDescription` um einen zweiten Quader erweitert, der aber als subtraktives Element (Attribut `isAddition`=false) definiert wird. Somit gilt diese Form als von der restlichen Geometrie „abgezogen“ - es entsteht ein Hohlraum. Das röhrenförmige Element in der Mitte des Legosteins kann auf ähnliche Weise modelliert werden. Ein Zylinder mit dem äußeren Durchmesser der Röhre wird als additives Element der `PartDescription` hinzugefügt, ein weiterer mit dem Innendurchmesser der Röhre als subtraktives Element. Die Position und Ausrichtung der einzelnen `Shape`s wird diesen über eine Transformationsmatrix (Klasse `Transformation`) über das Attribut `transformedBy` (dt.: umgewandelt durch) mitgegeben.

Parametrische 3D-CAD-Programme, wie beispielsweise PTC Creo [33], definieren ihre Geometrien auf ähnliche Weise. Ein Überführen der Geometrieinformationen aus einem solchen Programm wird somit erleichtert.

Die hier gezeigte Umsetzung ist bewusst sehr einfach gehalten. So ist zum Beispiel keine Schnittmenge von Geometrien vorgesehen und die Reihenfolge, in der die Vereinigungen



(a)



(b)

Abbildung 3.16: Geometrie eines klassischen „Vierer-Legosteins“

und Differenzen vorgenommen werden, muss über die Namen der Individuen festgelegt werden. Als Beispiele sind die Formen Zylinder (Klasse `Cylinder`) und Quader (Klasse `Box`) implementiert. Eine Geometrieabbildung, die beliebig komplexe Bauteile modellieren kann, ist nicht Teil dieser Arbeit. Allerdings zeigt sich hier einer der größten Vorteile der Verwendung einer Ontologie. Da diese strukturell offen ist, lassen sich spezialisiertere Ontologien verknüpfen, beispielsweise mit dem Ansatz von Anderson und Vasilakis [34], in dem Geometriedaten im STEP-Format [8] in einer Ontologie abgebildet sind.

3.5.2 Transformation

Um die modellierten `Shapes` im Raum zu positionieren, können Individuen der Klasse `Transformation` verwendet werden. Diese bilden einen Translationsvektor und eine Rotationsmatrix ab (Abbildung 3.15). Dies erlaubt es die Lage der `Shapes` in einem gemeinsamen Koordinatensystem festzulegen.

3.5.3 Ermittlung von Primitiven

Aus der Geometriebeschreibung ist es möglich, die Lage der Primitive eines Bauteils in einem gemeinsamen Koordinatensystem abzuleiten und diese für die Beschreibung von räumlichen Beziehungen bei Zusammenbauschritten zur Verfügung zu stellen. Allerdings ist dies nicht ganz trivial, da deren Lage im Raum durch mehrere Transformationen beeinflusst wird.

Transformationen werden, wie schon in Unterabschnitt 3.1.2 beschrieben, auch dazu verwendet, ganze Bauteile und Baugruppen zueinander zu positionieren. Da Bauteile in einem Produkt mehrmals verwendet werden können, ist die bei der Verwendung in einem `AssembleableElement` gespeicherte Transformation jedes mal verschieden. Für die Ableitung der Primitive muss also nicht nur die Lage der `Shapes` einer Bauteilbeschreibung berücksichtigt werden, sondern auch, wie dieser Bauteil im gesamten Produkt ausgerichtet ist. Werden Bauteile zu Baugruppen zusammengefasst, so werden auch diese mittels einer Transformation im Produkt positioniert. Für die Berechnung der Lage von Primitiven muss also jede Transformation einer Geometrie von der ersten Definition bis zur tatsächlichen Verwendung im Produkt nachvollzogen werden.

KAPITEL 4

Umsetzung der Ontologie

Dieses Kapitel stellt die für die Entwicklung verwendeten Werkzeuge vor und erläutert Details zur Umsetzung des Regelwerkes für die Erstellung der Planungsschritte. Um die Möglichkeiten der Ontologie testen zu können und gleichzeitig eine Beispielimplementierung für einen CAD-Import zu geben, wurde ein Algorithmus zum Import von Lego®-Modellen im LDraw-Dateiformat [35] implementiert. Dieser wird ebenso vorgestellt, wie Regeln zur Ableitung von Geometriedaten. Zuletzt werden eine Funktion zur Generierung einer Bauteilliste und ein Hilfswerkzeug zur einfachen Benutzung des Regelwerks gezeigt.

4.1 Werkzeuge zur Ontologieentwicklung

Für die Entwicklung und spätere Anwendung von Ontologien stehen diverse Werkzeuge zur Verfügung. Da eine Ontologie aufgrund beispielsweise von Änderungen der Domäne stets an neue Bedürfnisse angepasst können werden sollte [36], sind Werkzeuge, die nicht nur den Entwicklungsprozess, sondern auch die Anwendung ermöglichen, die bevorzugte Wahl. Nach [37] lassen sich Ontologie-Werkzeuge in folgende Kategorien einteilen:

- Entwicklungswerzeuge
- Werkzeuge für die Integration und Zusammenführung von Ontologien
- Bewertungswerkzeuge
- Kommentierungswerkzeuge
- Speicher- und Anfragewerkzeuge

Da am Institut an dem diese Arbeit entstanden ist (ACIN - Institut für Automatisierungs- und Regelungstechnik der TU Wien) das Werkzeug Protégé bereits für mehrere Projekte im Einsatz ist, empfiehlt sich dessen Verwendung. Protégé ist frei unter einer Open-Source-Lizenz verfügbar und erlaubt sowohl die Entwicklung als auch Kommentierung und Abfrage von Ontologien. Für die Erstellung von Inferenzregeln wurde die regelbasierte Programmiersprache Jess verwendet. Diese ist zwar kommerziell, bietet aber für den akademischen Gebrauch eine kostenlose Lizenz. Um Jess-Regeln und Code innerhalb der Protégé-Umgebung verwenden zu können, wird diese um das Tool JessTab erweitert. Für einige der Geometriefunktionen wurde auf Klassen der Bibliothek Java3D zurückgegriffen. Zusammengefasst wurden also folgende Werkzeuge verwendet:

- *Protégé* für die Entwicklung der Ontologie [38]
- *Jess* für die Entwicklung des Regelwerkes [39]
- *JessTab* für die Integration von Jess in die Protégé Umgebung (und vice versa) [40]
- *Java und Java3D* zur Erweiterung von Jess um einige geometrische Funktionen [41], [42]

Die Beschreibung der Werkzeuge im Folgenden soll einen kurzen Überblick geben. Nähere Informationen finden sich z.B. in [43] und der oben referenzierten Internetliteratur.

4.1.1 Protégé

Protégé ist „A free, open-source ontology editor and framework for building intelligent systems." [38] (dt. „Ein freier Open-Source-Ontologie-Editor und ein Framework zur Erstellung intelligenter Systeme.") Es stellt verschiedene Werkzeuge zur Erstellung eines Domänen-Modells, der Verwaltung von Individuen und zur Visualisierung von Ontologien bereit. Die Werkzeuge sind in sogenannten „Tabs" organisiert. Ein Tab stellt spezifische Möglichkeiten für den Umgang mit der Ontologie zur Verfügung, beispielsweise den Zugriff auf Attribute oder die Klassenhierarchie.

Protégé unterstützt zwei verschiedene Arten der Ontologie-Entwicklung. Über Protégé-Frames können frame-basierte Ontologien nach Open Knowledge Base Connectivity (OKBC) [44] entwickelt werden. Diese modellieren eine hierarchische Klassenstruktur, die Konzepte in einer Anwendungsdomäne repräsentieren. Sogenannte „Slots" definieren die Eigenschaften und Beziehungen der Klassen. Mit dem Editor Protégé-OWL können OWL-basierte Ontologien für Semantic-Web-Anwendungen entwickelt werden. [45] beschreibt unter anderem als Unterschied die Verwendung der „Open World Assumption" bei OWL an Stelle der „Closed World Assumption" bei framebasierten Ontologien (siehe auch Abschnitt 2.1.6)

Diese Arbeit verwendet die Variante Protégé-OWL (in Folge nur noch kurz Protégé genannt).

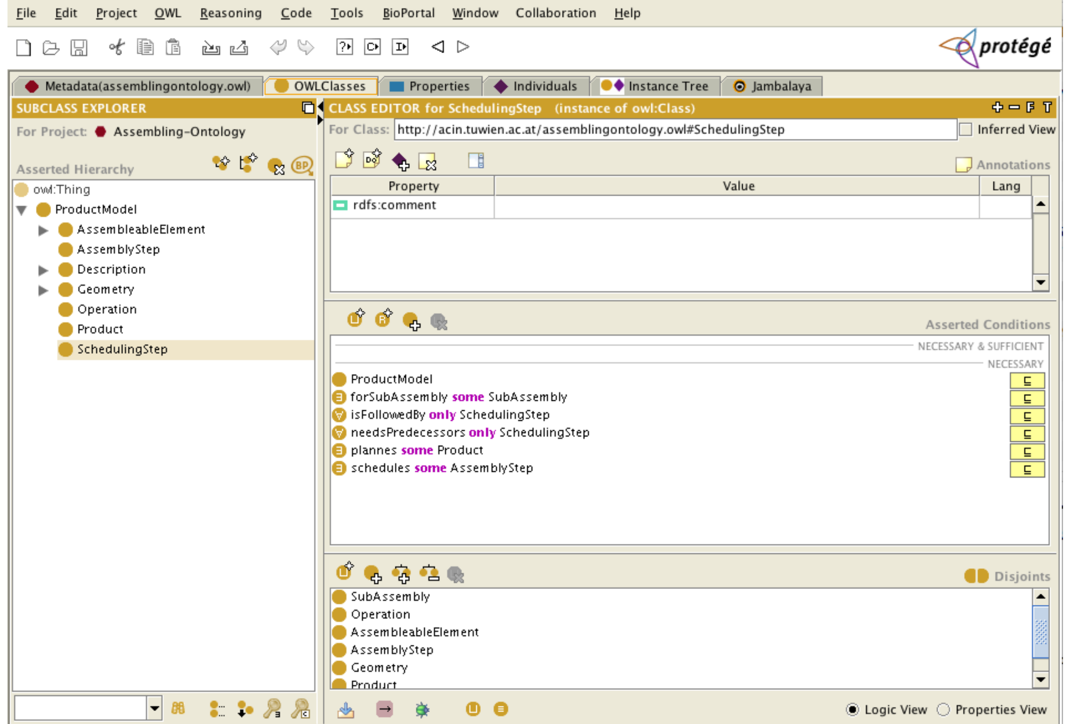


Abbildung 4.1: OWL-Klassenansicht in Protégé

Abbildung 4.1 zeigt die OWL-Klassenansicht. Im Subclass-Explorer auf der linken Seite kann die Klassenhierarchie bearbeitet werden. Im Class-Editor auf der rechten Seite können Klassen-Metadaten und Einschränkungen konfiguriert werden. Für die Formulierung der Einschränkungen stehen komfortable Assistenten zur Verfügung, die Hierarchie kann sehr einfach per Drag&Drop-Funktionalität verändert werden. Wird die Ansicht in den Properties-View (Auswahl rechts unten) geschaltet, so kann man sehr einfach Attribute und Properties zu Klassen hinzufügen.

Zur Visualisierung dient der Jambalaya-Tab (Abbildung 4.2). Dieser wird mit Protégé zwar mitgeliefert, muss aber erst aktiviert werden. In diesem Tab können Klassen, ihre Individuen und die Verbindungen untereinander auf vielfältige Weise angezeigt werden. Bei der Überprüfung ob die erstellten Ableitungsregeln richtig funktionieren, ist die Ansicht aber eine unschätzbare Hilfe, da Fehler wie fehlende oder überflüssige Beziehungen sichtbar werden.

4.1.2 Jess

Zur Entwicklung der Regeln für diese Ontologie wird die regelbasierte Programmiersprache Jess verwendet. Sie wurde von Ernest Friedman-Hill als internes Forschungsprojekt der Sandia National Laboratories entwickelt [39].

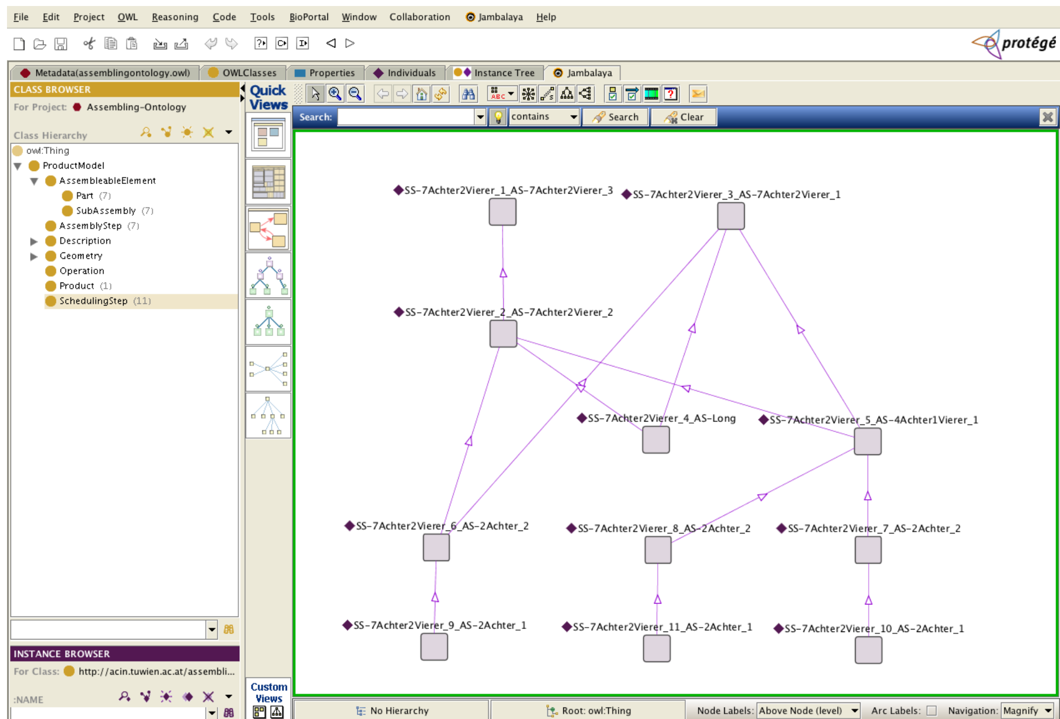


Abbildung 4.2: Jambalaya-Ansicht in Protégé mit mehreren Individuen und Beziehungen zwischen ihnen.

```
1 (defrule Regelname
2    (Muster)
3 =>
4    (Aktionen)
5    ; ein Kommentar
6 )
```

Listing 7: Skelett einer Regel in Jess

Die „Java Expert System Shell“ ist ein Werkzeug zur Erstellung von Expertensystemen. Ein Expertensystem besteht aus einer Inferenzmaschine, die wiederholt einen Regelsatz auf eine Menge von Fakten über ein Anwendungsgebiet anwendet. Eine Regel ist im Prinzip ein „Wenn-Dann“ Konstrukt. Die Inferenzmaschine überprüft nacheinander Fakten, ob diese auf das Muster im „Wenn“-Teil der Regel passen und führt in dem Fall die Aktionen im „Dann“-Teil aus (Listing 7) - dies wird auch als „feuern der Regel“ bezeichnet. Sind alle Fakten abgearbeitet, beginnt ein neuer Durchlauf und die Inferenzmaschine überprüft erneut alle Fakten.

Jess verwendet für diesen Prozess den sogenannten Rete-Algorithmus, der nach [46] deswegen schnell und effizient arbeiten kann, weil er Fakten nur mit den wahrscheinlichsten Regeln vergleicht. Die Geschwindigkeit geht allerdings zu Lasten des Speicherverbrauchs, da die Zwischenspeicherung von Vergleichsergebnissen zusätzlichen Speicher beansprucht.

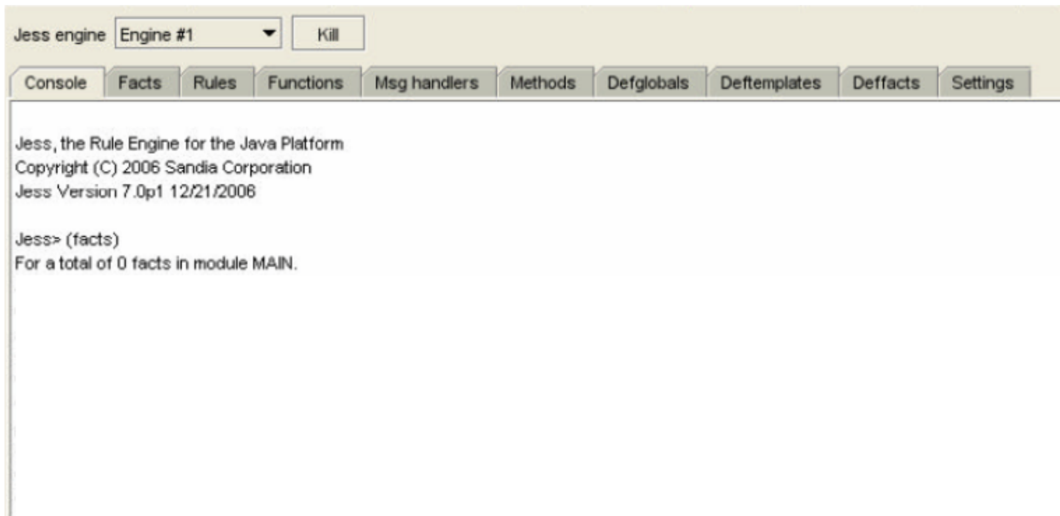


Abbildung 4.3: JessTab - Integration von Jess in Protégé

Jess ähnelt in seiner Syntax listenorientierten Programmiersprachen wie LISP [47].

Die Vorteile von Jess liegen in der Fähigkeit zur Integration von Klassen und Bibliotheken der Programmiersprache Java [41] in die Regelbasis. Damit lassen sich alle Möglichkeiten, die diese Programmiersprache zur Vefügung stellt, auch in Jess-Code nutzen. Umgekehrt kann die Inferenzmaschine sehr einfach in eigene Java-Programme integriert werden. Ein Beispiel dafür ist JessDE, eine Integration von Jess in die Programmierumgebung Eclipse [48] (Unterabschnitt 4.1.4).

In dieser Arbeit wird Jess vor allem dazu verwendet, den Zusammenbau entsprechend der in der Ontologie abgelegten Daten zu Produkten, deren Baugruppen und Fertigungsschritten zu planen. Dazu werden vom Regelwerk neue Fakten (in Form von Instanzen von Klassen) in der Ontologie abgelegt, die sich direkt aus den vorhandenen Daten ableiten (siehe dazu Abschnitt 3.3). Ändern sich die grundlegenden Fakten in der Ontologie, so kann das Regelwerk erneut darauf reagieren und den Zusammenbau umplanen.

4.1.3 JessTab

JessTab ist eine Erweiterung für Protégé, die es einerseits erlaubt Jess-Code direkt in einem Tab in Protégé zu schreiben und auszuführen (Abbildung 4.3). Andererseits wird Jess um Funktionen erweitert, die Protégé-Wissensbasen auf Jess-Fakten abbildet. Es werden auch Mechanismen zur Bearbeitung von Protégé-Daten aus Jess heraus angeboten [40].

Listing 8 zeigt ein Beispiel wie Daten zwischen Protégé und Jess ausgetauscht werden. Zuerst wird eine Klasse in die Faktenbasis übertragen (1). Danach wird ein Individuum (Instanz) dazu erstellt und dessen Attribute (Slots) mit Werten gefüllt (3). Eine Referenz

auf das Individuum wird in einer Variable gespeichert (2). Der Wert eines Attributes wird geändert (4) und schließlich werden beide Attribute ausgegeben (5).

```
1  (mapclass eineProtegeKlasse) ; (1)
2  (bind ?var ; (2)
3    (make-instance Instanzname of eineProtegeKlasse   ; (3)
4      (attribut1 "Hallo Welt")
5      (attribut2 42)
6      map
7    )
8  )
9  (slot-set ?var attribut1 "Hallo Welt!") ; (4)
10 (printout t (slot-get ?var attribut1) " - " (slot-get ?var attribut2)) ; (5)
```

Listing 8: Datenaustausch zwischen Protégé und JESS

4.1.4 JessDE - Jess Development Environment

Für die Entwicklung der Regeln wurde großteils die JessDE verwendet, das Jess in die Entwicklungsumgebung Eclipse integriert (Abbildung 4.4). Neben Syntax-Hervorhebungen für die Jess-Syntax wird der Code auch laufend auf Fehleingaben überprüft. Dateien mit Jess-Code können auf einfache Weise gestartet werden - im Hintergrund wird dazu automatisch eine Inferenzmaschine gestartet. Insgesamt gestaltet sich die Entwicklung mit der JessDE übersichtlicher als mit dem JessTab in Protégé. Eine Einschränkung von JessDE ist, dass die Syntaxerweiterungen von JessTab vom Syntax-Checker nicht unterstützt werden. Dadurch kommt es zu vielen falschen Fehlermeldungen und Warnungen im Code. Die Ausführung funktioniert dennoch.

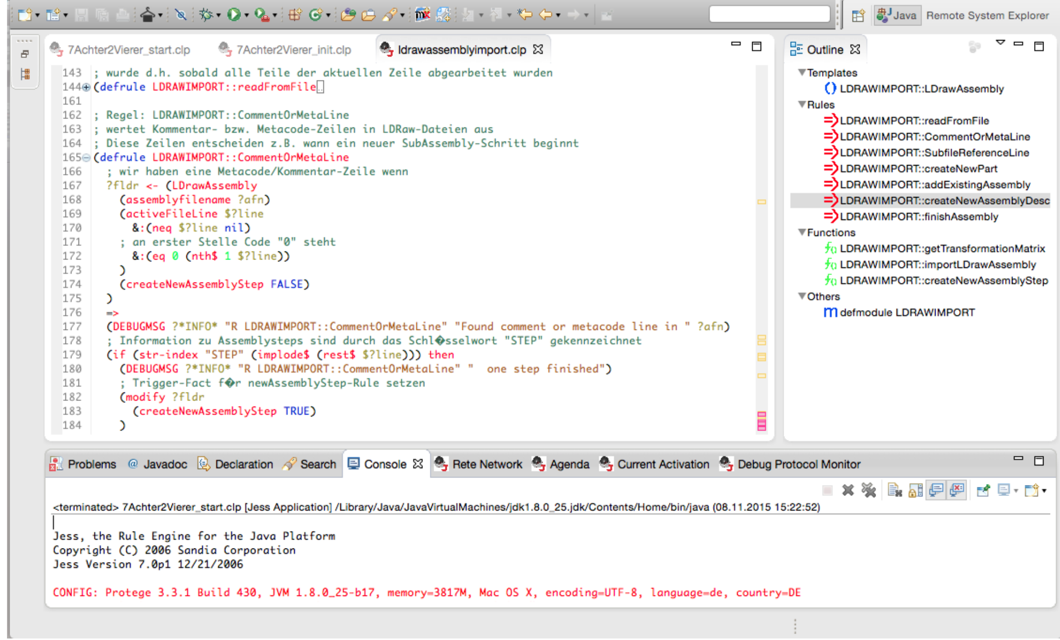


Abbildung 4.4: JessDE - eine Jess-Integration in Eclipse

4.2 Ableitungsregeln

Dieser Abschnitt soll einen Einblick in die Umsetzung der Ableitungsregeln geben. Die gezeigten Codeabschnitte sind ausführlich dokumentiert, Detailinformationen zur Vorgehensweise der Regeln und Funktionen können daher den Kommentarzeilen im Code entnommen werden.

4.2.1 Erzeugen einer Reihenfolge

Für die Ableitung der Reihenfolge aus den Individuen in der Ontologie ist im Wesentlichen die Regel *processAssemblyStepsIfTheyAreEndsteps* (Listing 9) zuständig. Sie erzeugt für jeden `AssemblyStep` einen `SchedulingStep`. Zuerst werden alle `AssemblySteps` verarbeitet, die keine Vorgänger für andere des selben `SubAssemblys` sind - also die letzten Arbeitsschritte einer Baugruppe sind. Diese werden dann aus der Faktenmenge der zu verarbeitenden Arbeitsschritte gelöscht, somit gibt es wiederum neue „letzte" Arbeitsschritte. Diese Regel feuert so lange, bis alle `AssemblySteps` abgearbeitet wurden.

```
1 (defrule SCHEDULEPRODUCT::processAssemblyStepsIfTheyAreEndsteps
2 ; es gibt ein 1. AS, das noch nicht verarbeitet ist
3   ?fastp <- (processedAssemblyStep
4     (processedProduct ?fldp)
5     (subassembly ?aktsa)
6     (assemblystep ?astoprocess)
7   )
8 ; es existiert aber KEIN unverarbeiteter 2. AS des selben SA, der den 1. AS als
      Vorgänger hat
9 ; -> der 1. AS ist ein Letzter in der Verarbeitungsreihenfolge eines SA
10  (not (processedAssemblyStep
11    (processedProduct ?fldp)
12    (subassembly ?aktsa)
13    (assemblystep ?as&:(member$ ?astoprocess (slot-get ?as needsPredecessors)))
14  ) )
15 ; vom zugehörigen processedProduct holen wir uns die Step-Nummer
16  ?fldp <- (processedProduct
17    (product ?product)
18    (productname ?productname)
19    (nextSchedulingStepNr ?psn)
20  )
21  =>
22 ; für den 1. AS erzeugen wir einen neuen SS
23  (mapinstance (make-instance (str-cat ?*PrefixSchedulingStep* ?productname "_" ?psn "_"
      (slot-get ?astoprocess :NAME)) of SchedulingStep
24      (plannes ?product)
25      (schedules ?astoprocess)
26      (forSubAssembly ?aktsa)
27      (isFollowedBy ())
28      (beginEarliest 0)
29      (finishLatest 0)
30      (stepnr ?psn)
31  ))
32 ; Fact zum 1. AS löschen, damit die Regel nicht mehr feuert
33  (retract ?fastp)
34 ; SS-Nummer erhöhen
35  (modify ?fldp (nextSchedulingStepNr (+ ?psn 1)))
36 )
```

Listing 9: Regel *processAssemblyStepsIfTheyAreEndsteps*

Damit dies überhaupt möglich wird müssen zuerst alle `AssemblyStep`-Individuen aller `SubAssemblys` des Produktes in die Jess-Faktenbasis abgebildet werden. Dies wird von

der Regel *createProcessingFactsForProduct* (Listing 20 im Anhang) durchgeführt.

Nach Abarbeitung dieser Regel ist jedem `AssemblyStep` ein `SchedulingStep` zugeordnet. Diese müssen jetzt noch korrekt miteinander in Beziehung gesetzt werden. Wie in Abschnitt 3.3 beschrieben, kümmern sich zwei Regeln um die Herstellung der Beziehungen. Die Regel *setSSPredecessorsSameSA* (Listing 10) verbindet alle `SchedulingSteps` einer Baugruppe miteinander (so es mehrere gibt) und die Regel *setSSPredecessorsDifferentSA* (Listing 11) setzt alle Beziehungen zwischen den `SchedulingSteps` verschiedener Baugruppen. Solange für jede Baugruppe nur ein `AssemblyStep` vorgesehen ist, oder alle `AssemblySteps` einer Baugruppe eine strikte Folge bilden, d.h. für jede Baugruppe gibt es genau einen ersten und einen letzten Bearbeitungsschritt, sind diese Verknüpfungen recht einfach zu ermitteln. Da komplexere Abläufe für den Zusammenbau aber ebenso erlaubt sind (siehe dazu beispielsweise Abbildung 3.12), bei denen es sowohl mehrere Anfangs- und/oder mehrere Endbearbeitungsschritte für jede `AssemblyDescription` geben kann, müssen in den Regeln einige Sonderfälle bedacht werden (siehe Kommentare in den Listings).

```
1  (defrule SCHEDULEPRODUCT::setSSPredecessorsSameSA
2  ; gibt es einen 1. SS
3    (object
4      (is-a SchedulingStep)
5      (OBJECT ?dependantstep)
6      (schedules ?dependantas)
7      (forSubAssembly ?dependantsa)
8    )
9  ; und einen 2. SS vom selben SA, der aber nicht schon Vorgänger des 1. SS ist (sonst
       Endlosschleife)
10   (object
11     (is-a SchedulingStep)
12     (OBJECT ?startstep)
13     (schedules ?startas)
14     (forSubAssembly ?dependantsa)
15     (needsPredecessors $?startpred&:(not (member$ ?dependantstep $?startpred)))
16   )
17 ; und gibt es einen zum 2. SS passenden AS, der den vom 1. SS verwalteten AS als
       Vorgänger hat
18   (object
19     (is-a AssemblyStep)
20     (OBJECT ?startas)
21     (needsPredecessors $?aspred&:(member$ ?dependantas $?aspred))
22   )
23   =>
24 ; dann "kopiere" die Abhängikeit der AS auf die SS
25   (slot-insert$ ?startstep needsPredecessors 1 ?dependantstep)
26   (slot-insert$ ?dependantstep isFollowedBy 1 ?startstep)
27 )
```

Listing 10: Regel *setSSPredecessorsSameSA*

```
1  (defrule SCHEDULEPRODUCT::setSSPredecessorsDifferentSA
2  ; gibt es einen 1. SS
3    (object
4      (is-a SchedulingStep)
5      (OBJECT ?dependantstep)
6      (:NAME ?dependantstepname)
7      (forSubAssembly ?dependantsa)
8    )
9  ; der selber kein Vorgänger im selben SA ist
10 ; (je nach Reihenfolge der Regelfeuerung kann es sein, dass das trotzdem passiert. Dafür
       gibt es aber eine Regel, die
```

```
11 ; das ausputzt. Damit jetzt nicht diese Regel und die Ausputzerregel im Wechsel feuern,
      muss es diese Bedingung geben)
12   (not (object
13     (is-a SchedulingStep)
14     (forSubAssembly ?dependantsa)
15     (needsPredecessors $?dependantpred&:(member$ ?dependantstep $?dependantpred))
16   ))
17 ; gibt es einen 2. SS der den ersten noch nicht als Vorgänger hat (Schutz vor
      Endlosfeuern)
18   (object
19     (is-a SchedulingStep)
20     (OBJECT ?startstep)
21     (:NAME ?startstepname)
22     (needsPredecessors $?predecessors&:(not (member$ ?dependantstep $?predecessors)))
23     (forSubAssembly ?startsa&:(neq ?startsa ?dependantsa))
24   )
25 ; gibt es ein zum 2. SS passendes SA
26   (object
27   (is-a SubAssembly)
28     (OBJECT ?startsa)
29     (applies ?startad)
30   )
31 ; gibt es zu diesem SA eine passende AD und enthält diese das SA vom 1. SS
32   (object
33     (is-a AssemblyDescription)
34     (OBJECT ?startad)
35     (isConstructedBy $?startae&:(member$ ?dependantsa $?startae))
36   )
37 ; und gibt es KEINEN anderen SS der das selbe SA bedient wie der 2. SS und schon
      Vorgänger des 2. SS ist
38 ; (dieses Pattern soll verhindern, das unnötige Vorgänger-Beziehungen angelegt werden.
      Hat ein SS einen Vorgänger im selben SA -
39 ; d.h. es gibt mehrere AS für eine AD mit gegenseitigen Abhängigkeiten - dann muss hier
      nicht unnötig auf die Schritte eines anderen SA
40 ; verwiesen werden.)
41 ; das OR verhindert auch, dass wenn es mehrere "gleiche" SS gibt (z.B. Subschritte von
      mehrfach verwendeten ADs), dass diese allen ihren
42 ; übergeordneten Schritten zugeordnet werden, sondern immer nur einer.
43   (not (object
44     (is-a SchedulingStep)
45     (OBJECT ?stepfromsamesa&:(member$ ?stepfromsamesa $?predecessors))
46     (forSubAssembly ?samesa&:(or (eq ?samesa ?startsa) (eq ?samesa ?dependantsa)))
47   ))
48 ; es darf auch KEINEN anderen SS geben, der ein anderes SA baut und den 1.SS für sich
      beansprucht. In Kombination mit der OR-Regel oben
49 ; lassen sich so mehrere "gleiche" SS aufteilen, sodass nicht einer bei allen
      weiterführenden Steps verlinkt wird.
50   (not (object
51     (is-a SchedulingStep)
52     (OBJECT ?doublestep&:(neq ?doublestep ?startstep))
53     (needsPredecessors $?doublepred&:(member$ ?dependantstep $?doublepred))
54     (forSubAssembly ?doublesa&:(neq ?doublesa ?startsa))
55   ))
56   =>
57 ; dann erzeuge eine Abhängigkeit zwischen 1. und 2. SS
58   (slot-insert$ ?startstep needsPredecessors 1 ?dependantstep)
59   (slot-insert$ ?dependantstep isFollowedBy 1 ?startstep)
60 )
```

Listing 11: Regel *setSSPredecessorsDifferentSA*

Ein weitere Komplexität ergibt sich dadurch, dass alle Regeln der Produktplanung zum selben Zeitpunkt aktiv sind. Es ist daher nicht vorhersehbar, in welcher Reihenfolge diese feuern. Im schlimmsten Fall werden falsche Beziehungen gesetzt, die anschließend identifiziert und wieder gelöst werden müssen. Dieser Fall tritt ein, wenn die Regel zur

Verknüpfung der `SchedulingSteps` zwischen verschiedenen Baugruppen bereits aktiv wird, obwohl noch nicht für alle `AssemblySteps` ein `SchedulingSteps` existiert. Dadurch feuert sie unter Umständen auf `SchedulingSteps`, die sich gar nicht auf Anfangs- bzw. Endbearbeitungsschritte beziehen, da diese Information in der Faktenbasis noch gar nicht vorhanden ist. Das Beispiel in Abbildung 4.5 soll dies verdeutlichen. Bildteil (a) zeigt eine Baugruppenstruktur, für die bereits einige, aber nicht alle `SchedulingSteps` erzeugt wurden (blaue Pfeile). Der Algorithmus der Regelmaschine entscheidet nun, jeweils einmal die Regel für baugruppeninterne Verlinkung (*setSSPredecessorsSameSA*, grüne Pfeile) und einmal jene für die Verlinkung zwischen verschiedenen Baugruppen (*setSSPredecessorsDifferentSA*, rote Pfeile) zu feuern - entsprechende Fakten liegen ja bereits in der Faktenbasis. Daraus ergibt sich die Situation in Bildteil (b). Nun wird der fehlende `SchedulingStep` ergänzt (Regel *processAssemblyStepsIfTheyAreEndsteps*). Da nun alle `SchedulingSteps` erzeugt sind, feuern nur mehr die Verlinkungsregeln und erzeugen die zusätzlichen korrekten Verlinkungen (c). Jetzt ist ersichtlich, dass vom vorherigen Schritt eine falsche Beziehung vorhanden ist.

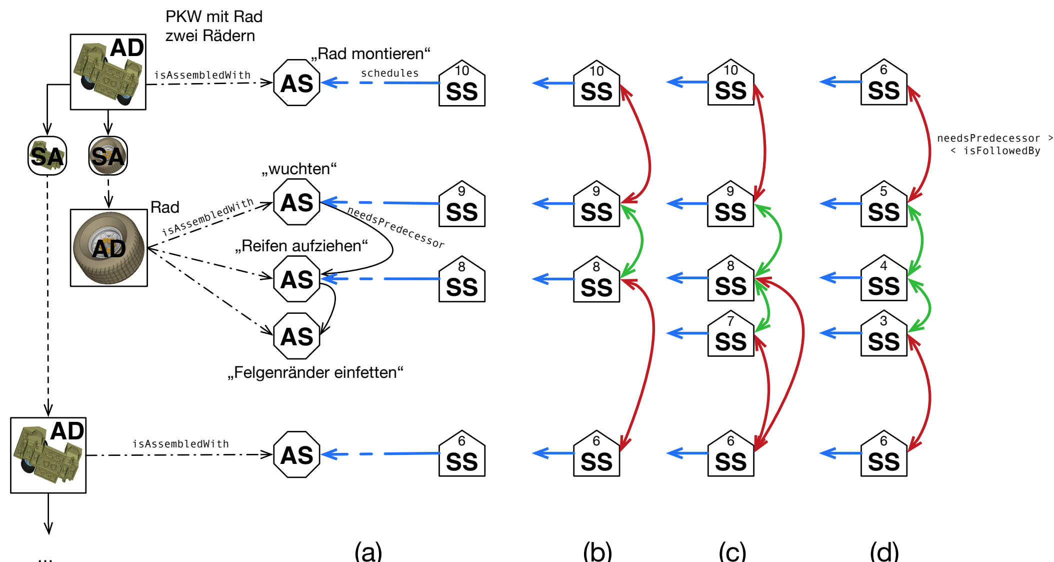


Abbildung 4.5: Unterschiedliche Reihenfolgen beim Feuern der Regeln kann zu falschen Ergebnissen führen.

Für die Beseitigung der falschen Beziehungen sind die folgenden Regeln zuständig:

- *removePredecessorFromDifferentSAifPredecessorFromSameSAexists* (Listing 12) ist sehr ähnlich und entfernt falsche Beziehungen von `SchedulingSteps`, die keine Anfangsschritte sind, aber dennoch Vorgänger in anderen Baugruppen haben. Diese Regel entfernt die falsche Beziehung in Abbildung 4.5 (c). Somit ergibt sich das Endergebnis, das in (d) zu sehen ist.

- *removeFollowerFromDifferentSAifFollowerFromSameSAexists* (Listing 21 im Anhang) entfernt falsche Beziehungen von `SchedulingSteps`, die keine Endschritte sind, aber dennoch Nachfolger in anderen Baugruppen haben.

```
1  (defrule SCHEDULEPRODUCT::removePredecessorFromDifferentSAifPredecessorFromSameSAexists
2  ; gibt es einen 1. SS
3    (object
4      (is-a SchedulingStep)
5      (OBJECT ?startstep)
6      (:NAME ?startstepname)
7      (forSubAssembly ?startsa)
8      (needsPredecessors $?startpred)
9    )
10 ; und einen 2. SS aus dem gleichen SA, der beim 1. SS Vorgänger ist
11   (object
12     (is-a SchedulingStep)
13     (OBJECT ?samesastep&:(member$ ?samesastep $?startpred))
14     (forSubAssembly ?startsa)
15   )
16 ; und gibt es einen 3. SS aus einem ANDEREN SA wie der 1. SS, der beim 1. SS Vorgänger
      ist
17   (object
18     (is-a SchedulingStep)
19     (OBJECT ?differentsastep&:(member$ ?differentsastep $?startpred))
20     (:NAME ?differentsastepname)
21     (forSubAssembly ?differentsa&:(neq ?startsa ?differentsa))
22     (isFollowedBy $?differentsafol)
23   )
24   =>
25 ; dann hat der SS aus dem GLEICHEN SA Vorrang und die andere Abhängigkeit wird gelöscht
26   (bind ?posp (member$ ?differentsastep $?startpred))
27   (bind ?posf (member$ ?startstep $?differentsafol))
28   (slot-delete$ ?startstep needsPredecessors ?posp ?posp)
29   (if (neq ?posf FALSE) then
30     (slot-delete$ ?differentsastep isFollowedBy ?posf ?posf)
31   )
32 )
```

Listing 12: Regel *removePredecessorFromDifferentSAifPredecessorFromSameSAexists*

4.2.2 Berechnung des frühestmöglichen Beginns und des spätestmöglichen Fertigstellungszeitpunktes

Listing 13 zeigt die Regel, die für die Berechnung des frühestmöglichen Beginns zuständig ist. Sie feuert auf jeden `SchedulingStep`, der Vorgänger hat und bei dem der Beginnzeitpunkt noch nicht gesetzt wurde. Zuerst wird ermittelt, wann alle Vorgängerschritte mit ihrer Arbeit fertig sind. Dazu dient die Funktion *getMaxPredecessorLevel* (Listing 14). Sie liefert die längste Dauer eines Vorgängers zurück. Dieser Wert wird dann als frühestmöglicher Beginn im `SchedulingStep` gesetzt.

```
1  (defrule SCHEDULEPRODUCT::setBeginEarliest
2    (object
3      (is-a SchedulingStep) ; suche einen SS
4      (OBJECT ?step)
5      (:NAME ?stepname)
6      (beginEarliest ?level&:(and (neq ?level nil) (eq ?level 0))) ; dessen Level noch
      nicht berechnet wurde (nil oder 0)
7      (isFollowedBy $?successors)
8      (needsPredecessors $?predecessors&:(> (length$ $?predecessors) 0)) ; und der
      Vorgänger hat.
9    )
```

```
10   =>
11   (bind ?newlevel (SCHEDULEPRODUCT::getMaxPredecessorLevel ?step)) ; ermittle den
       spätesten Zeitpunkt, an dem die Vorgänger fertig sind
12   (if (neq ?level ?newlevel) then ; verhindert unnötiges Verändern der Fakts, was zu
       unnötigen Regelaufrufen führen würde
13     (slot-set ?step beginEarliest ?newlevel) ; den neuen Beginnzeitpunkt setzen
14     ; Level bei allen Nachfolgern auf 0 setzen um die Berechnung neu anzustoßen
15     (foreach ?successor $?successors
16       (slot-set ?successor beginEarliest 0)
17     )
18   )
19 )
```

Listing 13: Regel *setBeginEarliest*

```
1  (deffunction SCHEDULEPRODUCT::getMaxPredecessorLevel(?SchedulingStep)
2  ; Sucht den maximalen Level eines Vorgängerarbeitsschrittes
3  ;   Rückgabewerte
4  ;   - maximaler Level aller vorher notwendigen Schritte
5  ;   - wenn keine Schritte vorher notwendig sind -> 0
6  ;   - wenn ein vorheriger Arbeitsschritt noch keinen Level gesetzt hat -> -1
7
8  ; alle Vorgänger des übergebenen SS holen
9    (bind $?predecessors (slot-get ?SchedulingStep needsPredecessors))
10 ; es gibt keine Vorgängerschritte -> 0 zurückgeben
11   (if (eq (length$ $?predecessors) 0) then
12     (return 0)
13   )
14   (bind ?maxlevel -1)
15   (foreach ?predecessor $?predecessors
16     (bind ?level (slot-get ?predecessor beginEarliest))
17     (if (eq ?level nil) then
18       (return -1)
19     )
20 ; der frühestmögliche Fertigstellunszeitpunkt ist der frühestmögliche Beginn
21 ; + die geplante Dauer. Das Maximum wird in maxlevel gespeichert.
22     (bind ?level (+ ?level (SCHEDULEPRODUCT::getDuration ?predecessor)))
23     (if (> ?level ?maxlevel) then
24       (bind ?maxlevel ?level)
25     )
26   )
27   (return ?maxlevel)
28 )
```

Listing 14: Regel *getMaxPredecessorLevel*

Für den spätestmöglichen Fertigstellungszeitpunkt beginnt die Regel *setFinishLatest-ForFinishingSteps* damit, für jeden `SchedulingStep` der keine Nachfolger hat, einen Fertigstellungszeitpunkt zu setzen (Listing 15). Danach wird für alle Vorgänger der Fertigstellungszeitpunkt auf 0 gesetzt, wodurch für diese der spätestmögliche Fertigstellungszeitpunkt neu berechnet wird. Gibt es mehrere solche letzte Schritte - weil z.B. in der produktzugehörigen `AssemblyDescription` mehrere parallele Schritte definiert sind - dann wird der spätestmögliche Zeitpunkt aller letzten Schritte für alle gesetzt. Dies wird mit der Regel *correctFinishLatestForFinalSA* im Nachhinein für alle letzten Schritte gesetzt (Listing 22 im Anhang).

```
1 (defrule SCHEDULEPRODUCT::setFinishLatestForFinishingSteps
2   (object
3     (is-a SchedulingStep) ; suche einen SS
4     (OBJECT ?step)
5     (:NAME ?stepname)
```

```
6       (beginEarliest ?elevel&:(neq ?elevel nil)) ; der bereits einen Beginnzeitpunkt
          gesetzt hat
7       (finishLatest ?llevel&:(neq ?llevel nil)&:(<= ?llevel ?elevel)) ; der
          Fertigstellungszeitpunkt vor dem Beginnzeitpunkt liegt (das ist ein Zeichen dafür,
          dass der Fertigstellungszeitpunk noch nicht berechnet ist)
8       (isFollowedBy $?successors&:(eq (length$ $?successors) 0)) ; der keine Nachfolger
          hat (d.h. ein letzter Schritt bei der Produkterstellung)
9       (needsPredecessors $?predecessors)
10    )
11    =>
12    (bind ?duration (SCHEDULEPRODUCT::getDuration ?step)) ; die Dauer des Arbeitsschrittes
          aus dem verknüpften AS auslesen
13    (slot-set ?step finishLatest (+ ?elevel ?duration)) ; Fertigstellungszeitpunkt ist für
          Endschritte immer Beginn + Dauer
14  ; Gibt es mehrere FinishingSteps, so muss der Wert noch auf den größter aller
          FinishingSteps korrigiert werden. Siehe Regel correctFinishLatestForFinalSA
15  ; allen Vorgängern setzen wir den Fertigstellungszeitpunkt zurück, damit diese neu
          berechnet werden. Dabei dürfen aber die noch nicht fertig angelegten nicht
          verwendet werden
16    (foreach ?predecessor $?predecessors
17      (if (neq (slot-get ?predecessor finishLatest) nil) then
18        (slot-set ?predecessor finishLatest 0)
19      )
20    )
21  )
```

Listing 15: Regel *setFinishLatestForFinishingSteps*

Sind die Fertigstellungszeitpunkte für die Endschritte gesetzt, kann mit der Abarbeitung der weiteren Schritte begonnen werden. Die Planungsschritte werden dabei vom spätesten zum frühesten durchlaufen. Für jeden Schritt wird der frühestmögliche Beginn aller seiner Nachfolger ermittelt. Zu diesem Zeitpunkt muss der Vorgänger dann spätestens fertig sein. Listing 23 (im Anhang) zeigt die Regel *setFinishLatestLowerThanSuccessor*, die genau diese Funktion erfüllt.

4.3 CAD-Import - Eine Beispielimplementierung

Um Produkte zu beschreiben, muss die Information über die beteiligten Komponenten die zusammengebaut werden erst in die Ontologie eingepflegt werden. Dies kann manuell erledigt werden - beispielsweise mit dem für diese Arbeit verwendeten Werkzeug Protégé - oder diese Daten aus anderen Quellen importieren. Beispiele für sinnvolle Quellen wären etwa CAD-Baugruppen oder Teilekataloge.

Im Folgenden soll eine Beispielimplementierung zeigen, wie ein solcher Import umgesetzt werden kann. Dazu werden Zusammenbaudaten für Lego®-Modelle ausgelesen und entsprechend Teile- und Baugruppenbeschreibungen, Bauteilgruppen und Arbeitsschritte erzeugt. Lego wurde aus verschiedenen Gründen ausgewählt:

- Es existieren mehrere Lego-CAD-Programme, die es erlauben, Lego-Modelle zu designen und eine Aufbaureihenfolge festzulegen. Viele Programme speichern in einem gemeinsamen Dateiformat „.ldraw“. Für diese Arbeit wurde die frei verfügbare Software „Bricksmith“ [49] verwendet.

- Es existiert eine frei verfügbare Bauteilbibliothek [35], die in vielen Lego-CAD-Programmen wie Bricksmith eingesetzt werden kann und Teilenummern sowie Geometriedaten zu den einzelnen Bausteinen enthält.
- Lego wird mit „Zusammenbauen“ assoziiert, die erzeugten Baugruppen sind intuitiv begreifbar. Das vereinfacht eine Verifikation der Ergebnisse bei der Erstellung der Ableitungsregeln.

Die Benennung der erzeugten Individuen erfolgt nach dem Schema „x-y_z“, wobei

- x die Abkürzung des Klassennamens des Individuums ist
(z.B. AD für `AssemblyDescription`),
- y der Name des Modells ist (gibt es Sub-Modelle werden auch deren Namen verwendet) und
- z eine fortlaufende Nummer ist.

Das LDraw-Dateiformat ist ein Textformat, das sich aus einzelnen Befehlszeilen zusammensetzt. LDraw-Dateien können aufeinander referenzieren und benutzen diese Funktion sehr intensiv um komplexe Formen aus primitiven zusammenzusetzen. Es wird zwischen zwei Dateiendungen unterschieden, wobei die Syntax innerhalb bei beiden ident ist, sie aber verschiedenen Zwecken dienen:

- *.DAT*: Dateien mit dieser Endung enthalten Lego®-Bauteile. Diese sind aus mehreren Primitiven wie Linien, Dreiecken und Vierecken zusammengesetzt. Das LDraw-Format verwendet ein Flächenmodell um die Teile zu modellieren. Auch hier wird intensiv von Referenzierungen zwischen Bauteil-Dateien Gebrauch gemacht (Abbildung 4.6).
Da praxisrelevante 3D-CAD-Daten oft in einem Volumensmodell gespeichert sind, wurden Geometriedaten aus .DAT-Dateien nicht importiert, sondern aus den Referenzen auf Bauteile lediglich die Bauteilnummern extrahiert. Abschnitt 3.5 beschreibt eine Möglichkeit, Bauteilgeometrien mit Hilfe eines Volumensmodells in der Ontologie abzubilden. Wird ein Lego®-Baustein damit modelliert, muss dieser nach dem Schema „PD-ldr-*LDraw-Nummer*“ benannt werden, um vom Importalgorithmus berücksichtigt zu werden. Als Beispiel zeigt Abbildung 3.16 einen Baustein, der in der LDraw-Bibliothek unter der Nummer 3003 abgelegt ist. Die richtige Benennung der `PartDescription` für diesen Baustein ist also „PD-ldr-3001“.
- *.LDR*: Diese Dateien enthalten Modelle aus mehreren Bauteilen. Sie bestehen fast ausschließlich aus Referenzen auf Bauteildateien (.DAT) oder andere Sub-Modelle. Werden beim Import in einem Modell Primitive gefunden, werden diese ignoriert. Einzelne Zusammenbauschritte können über Bemerkungszeilen festgelegt werden. Ein Beispiel für eine .LDR-Datei zeigt Listing 16.

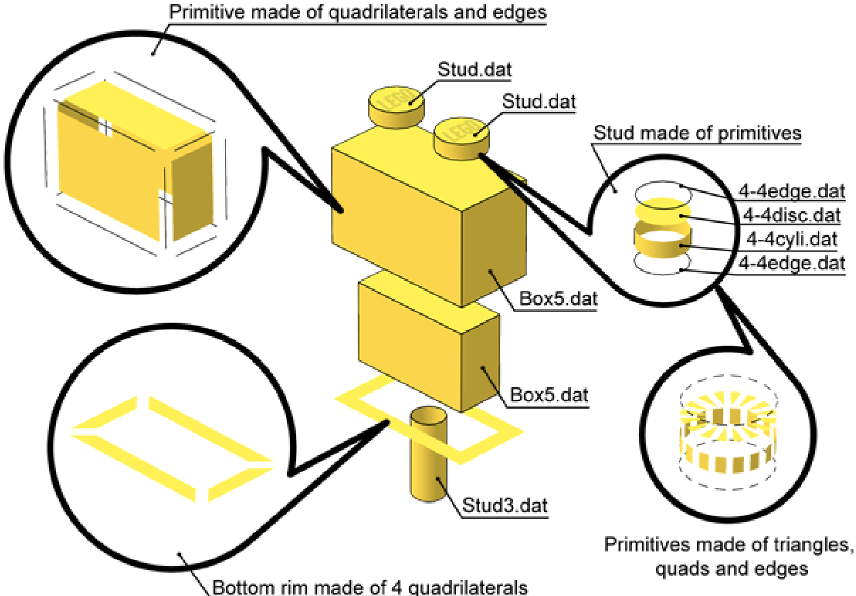


Abbildung 4.6: Geometrische Komponenten eines 1x2-Legosteins im Flächenmodell des LDraw-Formats (Bild: Holly-Wood.it [50])

Detailliertere Informationen zum Aufbau der Dateiformate finden sich unter [35]. Im Folgenden wird nur ein Überblick über die für den Import relevante Syntax geboten.

Die erste Ziffer bezeichnet den Typ der Zeile: 0 ist ein Kommentar oder eine Metainformation. Dort können Informationen wie Name des Modells oder der Autor hinterlegt sein. Für den Import relevant ist die Metainformation „STEP“. Sie kennzeichnet, dass ein neuer Zusammenbauschritt begonnen wird. Das bedeutet, dass in einer .LDR-Datei nicht nur die Modellgeometrien gespeichert sind, sondern auch Hinweise in welchen Schritten diese aufzubauen sind.

Eine 1 an erster Stelle markiert eine Referenzzeile. Für die Importfunktion sind nur diese beiden Typen relevant, alle anderen Ziffern werden ignoriert.

Der weitere Aufbau einer Referenzzeile setzt sich aus mehreren Datenfeldern zusammen (Listing 16):

- Die zweite Zahl (blaue Schrift im Listing) gibt die Farbe des referenzierten Bauteils bzw. der Baugruppe an. Die Nummer entspricht dabei einem Eintrag in einer LDraw-spezifischen Farbtabelle für Lego®-Farben.

- Die Zahlen an Position 3-5 (rote Schrift im Listing) definieren die Position des Bauteils im Modell-Koordinatensystem. Die verwendeten LDU-Einheiten richten sich nach üblichen Längen von Lego. 1 LDU entspricht 0,4mm.

```
1 0 Test-Modell
2 0 Name: test.ldr
3 0 Author: Erhard List
4 1 2 0    0  0   1        0 0   0 1 0  0    0 1        3001.DAT
5 1 1 0  -24  0   0        0 1   0 1 0  1    0 0        3001.DAT
6 1 4 23.5 -48 33.5 0.866025 0 0.5 0 1 0 -0.5 0 0.866025 3003.DAT
```

Listing 16: Aufbau einer LDR-Datei für das Modell in Abbildung 4.7

- Die Zahlen an Position 6-14 (grüne Schrift im Listing) definieren eine 3x3 Transformationsmatrix mit der eine Drehung bzw. Verzerrung des Teils definiert wird.
- Die letzte Stelle einer Zeile beinhaltet eine Referenz auf den verwendeten Bauteil (.DAT-Datei) oder ein anderes Modell (.LDR-Datei).

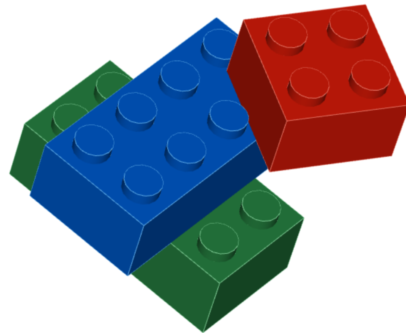


Abbildung 4.7: Test-Modell aus Listing 16. Der rote Baustein ist gegenüber dem darunterliegenden blauen um 30 Grad gedreht.

Die Importfunktion liest eine Modelldatei Schritt für Schritt ein und übersetzt die darin enthaltenen Daten in Individuen in der Ontologie. Das Modellkoordinatensystem und die Transformationen (Verschiebungen, Drehungen und Verzerrungen) werden ebenfalls importiert. Daraus können dann wie in Abschnitt 3.5 beschrieben für den Zusammenbau relevante Geometrien abgeleitet werden.

Exemplarisch zeigt Listing 17 die Verarbeitung einer Referenzzeile.

```
1  (defrule LDRAWIMPORT::subfileReferenceLine
2  ; Regel: LDRAWIMPORT::SubfileReferenceLine
3  ; wertet Zeilen mit Referenzen auf andere Bauteile/Assemblies aus
4  ; aber nur dann, wenn ein aktives SubAssembly besteht.
5    ?fldr <- (LDrawAssembly
6      ; Fakt, das die aktuell eingelesene Zeile aufgespaltet in Listenelemente enthält
7    (activeFileLine $?line
8        &:(neq $?line nil)
9        ; wir haben eine Referenz-Zeile wenn an erster Stelle Code "1" steht
10       &:(eq 1 (nth$ 1 $?line))
11     )
```

```
12      ; es muss ein SubAssembly bereits bestehen
13      ; (es darf kein SubAssembly—Erstellungsprozess gerade in Gang sein)
14      (createNewAssemblyStep FALSE)
15      (createNewPart FALSE)
16      (addExistingAssembly FALSE)
17    )
18    =>
19    (bind ?partfile (nth$ (length$ $?line) $?line))
20    ; Dateiendung herauslesen
21    ; LDraw—Dateien unterscheiden 3 Referenzen: ohne Endung, DAT und LDR)
22    (bind ?ppos (str—index "." ?partfile))
23    (if (eq ?ppos FALSE) then
24    ; Referenzen ohne Endung, also innerhalb einer Datei werden nicht unterstützt
25      ; restliche aktive Zeile leeren, damit nächste Zeile eingelesen wird
26      (modify ?fldr (activeFileLine (list)))
27     else
28       (bind ?ext (sub—string (+ ?ppos 1) (str—length ?partfile) ?partfile))
29    ; Namen des Individuums zusammenbauen. Der Prefix "ldr—" ist leider notwendig, da
         Protege keine numerischen Klassennamen erlaubt
30       (bind ?partclass (str—cat "ldr—" (sub—string 1 (— ?ppos 1) ?partfile)))
31    ; Transformatinsmatrix aus der Zeile lesen
32       (modify ?fldr (transformation (LDRAWIMPORT::getTransformationMatrix $?line)))
33       (if (eq (upcase ?ext) "DAT") then
34    ;    2. Filetype .dat —> vordefinierter Part
35    ;        —> wenn es zu diesem Teil bereits eine PD gibt —> Part anlegen, ansonsten
         zuerst eine PD anlegen und dann einen Part.
36       (modify ?fldr (createNewPart ?partclass))
37       )
38       (if (eq (upcase ?ext) "LDR") then
39    ;    3. Filetype .ldr —> anderes SubAssembly
40         (modify ?fldr (addExistingAssembly ?partfile))
41       )
42    )
43  )
```

Listing 17: Regel *subfileReferenceLine*

4.4 Erzeugung von Primitiven zur Definition räumlicher Beziehungen

Sind für Bauteile Geometriedaten hinterlegt, so können daraus Primitive wie Punkte, Kanten und Flächen extrahiert werden, um für den Zusammenbau relevante räumliche Beziehungen zu beschreiben (siehe auch Abschnitt 3.5). In diesem Abschnitt wird dazu eine Beispielimplementierung gezeigt. Diese erhebt keinen Anspruch auf Vollständigkeit zur Beschreibung beliebig komplexer Geometrien. Vielmehr soll sie das Konzept dahinter verdeutlichen, daher wird nur die Form eines Quaders als Grundlage verwendet. Die Vorgehensweise setzt sich aus den folgenden Schritten zusammen:

- Auswahl eines `AssembleableElement`, für dessen Fügung Primitive benötigt werden. Da dies sehr aufwändig ist, werden sie nur für die in diesem `AssembleableElement` enthaltenen Geometrien erstellt und nicht alle möglichen des gesamten Produktes.
- Ermittlung aller `Shapes`, die für das `AssembleableElement` relevant sind. Dazu muss die Struktur des Produktes vom `AssembleableElement` bis zu allen referenzierten `PartDescriptions` analysiert werden.

- Für alle gefunden `PartDescriptions` wird geprüft, ob dafür Geometrieinformationen vorhanden sind. Für diese werden anschließend alle Primitive als Individuen der entsprechenden Klassen `Vertex`, `Edge` oder `Area` erzeugt. Beispielsweise sind dies für einen Quader acht Punkte, zwölf Kanten und sechs Flächen (Listing 37 im Anhang).

- Doppelte Geometrien wie zusammenfallende Punkte, werden zusammengefasst, um die Anzahl der Individuen zu reduzieren.

- Auf jedes Element müssen nun alle Transformationen von der definierenden `PartDescription` bis zum betrachteten `AssembleableElement` angewendet werden. Dies wird hauptsächlich von der Regel *getAndTransformPrimitives* (Listing 18) durchgeführt. Einer der Vorteile der verwendeten Regelsprache Jess ist, dass Bibliotheken der Programmiersprache Java genutzt werden können. Dies wird hier genutzt, indem es die Bibliothek Java3d [42] einsetzt, um die Transformationen durchzuführen.

Das Ergebnis sind Primitive, die für das gewünschte `AssembleableElement` richtig positioniert im Produkt-Koordinatensystem liegen.

```
1  (defrule GEOMETRY::getAndTransformPrimitives
2    ?fpc <- (GEOMETRY::PrimitiveContainer
3      (owner ?owner)
4      (activeAE ?ae)
5      (primitives $?prim &:(eq (length$ $?prim) 0))
6      (activetransformation ?trans &:(neq ?trans nil))
7    )
8    (object
9      (is-a ?t1 &:(isClassOrSubclassOf ?t1 AssembleableElement))
10     (OBJECT ?ae)
11     (:NAME ?aename)
12     (applies ?desc)
13   )
14   (object
15     (is-a ?t2 &:(isClassOrSubclassOf ?t2 Description))
16     (OBJECT ?desc)
17     (isConstructedBy $?subelements &:(neq (length$ $?subelements) 0)) ; können wieder
     AEs oder Shapes sein. Ist leer, wenn der Teil keine Geometrie besitzt.
18   )
19   =>
20   (foreach ?sub $?subelements
21     (if (isClassOrSubclassOf (class ?sub) AssembleableElement) then
22     ; bei einem Assembleable-Element starten wir einen neuen Verarbeitungsstrang
23       ; die Primitive werden bei den Shapes dann dem ?owner zugewiesen. Wir können
     dieses Fact also löschen (sonst Endlosschleife!)
24     (retract ?fpc)
25       (call ?trans mul (JAVA3D::convPTransformation_JTransform3D (slot-get ?sub
     positionedWith)))
26       (assert (PrimitiveContainer
27         (owner ?owner)
28         (activeAE ?sub)
29         (primitives (create$))
30         (activetransformation ?trans) ; Achtung: Reihenfolge der Parameter von
     concatTransformation ist wichtig!
31       ))
32      else
33       (if (isClassOrSubclassOf (class ?sub) Shape) then
34       ; was anderes dürfte es ja gar nicht sein, aber Kontrolle ist besser
35         (bind $?primitives (slot-get ?sub hasGeometry))
36         (if (and (neq $?primitives nil) (neq (length$ $?primitives) 0)) then
```

```
37          (bind $?transformedprimitives (JAVA3D::JTransformPrimitivesToDuplicate ?owner ?
        trans $?primitives))
38        )
39        ; Shapes sind immer die untersten Elemente im Assemblybaum -> wir Kennzeichnen
        den PrimitivContainer als Blatt
40        ; da bereits beim duplizieren der Primitive das "belongsTo-Feld" gesetzt wird,
        können wir hier einfach aufhören und das Fact löschen
41        (retract ?fpc)
42      )
43    )
44  )
45 )
```

Listing 18: Regel *getAndTransformPrimitives*

4.5 Generierung einer Bauteilliste

Die Ontologie wurde auch unter dem Gesichtspunkt erstellt, die notwendigen Individuen für Bauteile und Baugruppen möglichst redundanzfrei abbilden zu können. Beschreibungen von in einem Produkt oft verwendeten Bauteilen sind somit nur einmal gespeichert, aber sehr oft referenziert. In der Ontologie ist somit zwar implizit gespeichert, wie viele Teile eines Typs für ein Produkt notwendig sind, dies lässt sich aber nicht direkt herauslesen.

Um dennoch eine Bauteilliste aus der Ontologie zu erhalten, wurden Funktionen in Jess formuliert, welche die richtige Anzahl an verwendeten Bauteilen aus der Ontologie ermitteln und in Form einer Liste zurückgeben. Listing 19 zeigt die zentrale Funktion *getPartsOfAssemblyDescription*. Diese ermittelt alle `Descriptions` die das `SubAssembly` beschreiben. Für alle `PartDescriptions` wird mittels der Funktion *partlistInsert* (Listing 24 im Anhang) die Bauteilliste erweitert. Entweder ist der Bauteil bereits auf der Liste, dann wird nur die Anzahl erhöht, oder der Bauteil wird neu der Liste hinzugefügt. Für alle `AssemblyDescriptions` wird die Funktion *getPartsOfAssemblyDescription* rekursiv aufgerufen. Da das „Ende“ einer logischen Bauteilhierarchie stets Individuen der Klasse `PartDescription` sind, endet die Rekursion. Abbildung 4.8 zeigt ein Produkt und die dazu erzeugte Bauteilliste.

```
1  (deffunction getPartsOfAssemblyDescription (?assembly ?partlist)
2  ; für alle Descriptions aus denen ein SA aufgebaut ist
3    (foreach ?p (slot-get ?assembly isConstructedBy)
4      (bind ?p (slot-get ?p applies))
5  ; ermittle den Typ des Individuums (seine Klasse)
6      (bind ?partclass (class ?p))
7      (if (eq (isClassOrSubclassOf ?partclass PartDescription) TRUE) then
8  ; ist es eine PD, dann wird die Bauteilliste erweitert
9        (bind ?pname (slot-get ?p :NAME))
10       (bind ?partlist (partlistInsert ?pname ?partlist))
11     else
12 ; sonst ist es ein SA. Für dieses wird die Funktion rekursiv aufgerufen.
13       (bind ?partlist (getPartsOfAssemblyDescription ?p ?partlist))
14     )
15   )
16   (return ?partlist)
17 )
```

Listing 19: Funktion *getPartsOfAssemblyDescription*

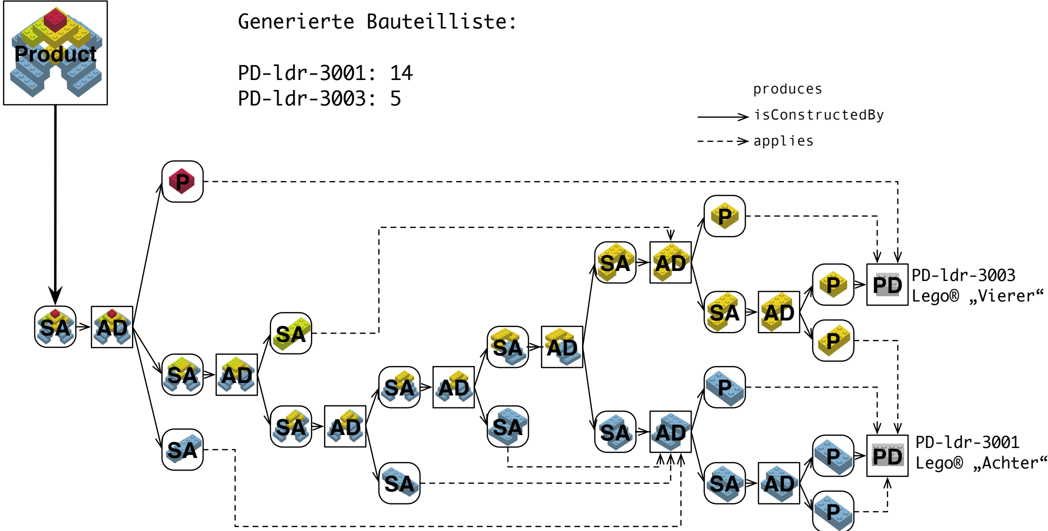


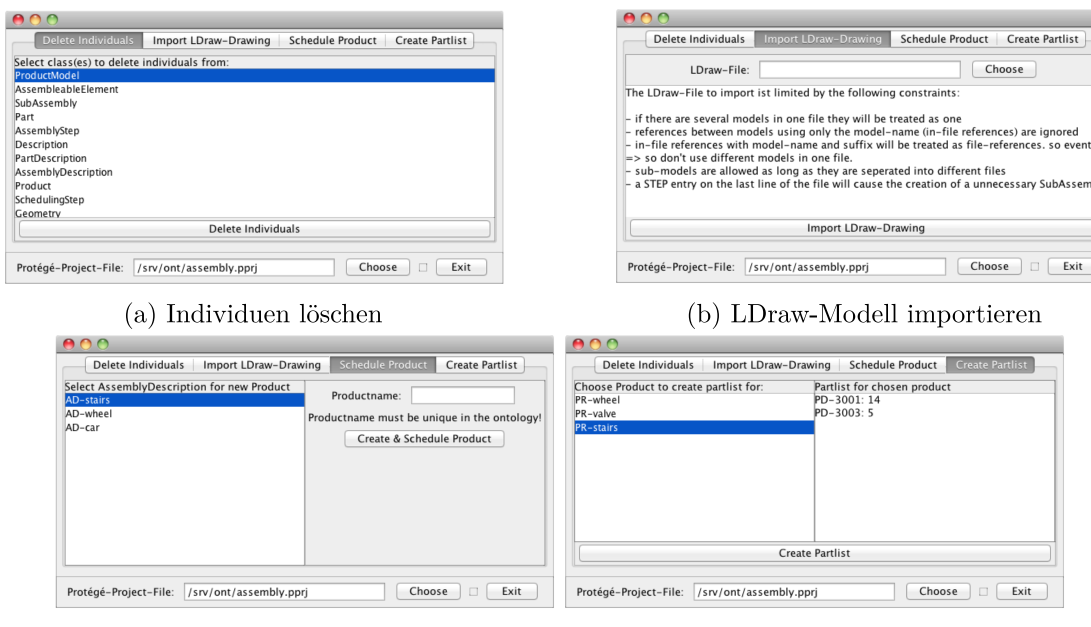
Abbildung 4.8: Bauteilhierarchie eines Produktes und die dazu erzeugte Bauteilliste

4.6 Hilfswerkzeug zur Nutzung der Regelbasis

Zusätzlich zur Möglichkeit, die Jess Regelbasis in JessTab von Protégé zu verwenden, wurde für die einfache Verwendung der Regelbasis eine Java-Applikation erstellt. Diese soll die Arbeit mit der Ontologie in Protégé nicht ersetzen, sondern ergänzen. Die Applikation bietet folgende Möglichkeiten:

- Import einer LDraw-Datei und Erzeugung eines Produktes dazu in der Ontologie. Für die einzelnen verwendeten Bauteile werden keine Geometriedaten importiert. Sind diese allerdings schon vorher in der Ontologie enthalten, werden sie verwendet.
- Erzeugung aller Planungsschritte für ein Produkt
- Generierung und Anzeige einer Bauteilliste für ein Produkt
- Löschen aller oder bestimmter Individuen aus der Ontologie. Dies ist eine Komfortfunktion um bei Tests Konflikte mit doppelten Benennungen von Individuen zu verhindern.

Die Ontologie wird aus einer Protégé-Projekt-Datei importiert. Die erzeugten Individuen werden in diese wiederum zurückgespeichert. Abbildung 4.9 zeigt Screenshots der Applikation.



(a) Individuen löschen

(b) LDraw-Modell importieren

(c) Zusammenbau planen

(d) Bauteilliste erzeugen

Abbildung 4.9: Java-Applikation zur einfachen Handhabung der Jess-Regelbasis für die Zusammenbau-Ontologie

KAPITEL 5

Ergebnisse

5.1 Diskussion der Ergebnisse

Mit dieser Arbeit wurde das Ziel angestrebt mit Hilfe einer Ontologie Wissen über den Zusammenbau von Produkten in maschinenlesbarer Form zur Verfügung zu stellen. Dies soll vor allem flexiblen Fertigungsanlagen Informationen liefern, um sich an verschiedene Produkte ohne manuelle Rekonfiguration der Anlage anzupassen. Im Folgenden wird das Ergebnis der Arbeit in Bezug auf dieses Ziel diskutiert.

Die Organisation der Bauteilstruktur in beschreibende Elemente (`PartDescription` und `AssemblyDescription`) und deren Anwendung in Form der `AssembleableElement`s zeigt sich bei Tests mit mehreren verschieden komplexen Produkten, die aus mehreren/vielen gleichen Bauteilen bzw. Baugruppen bestehen als vorteilhaft. Dies wären vor allem Produkte aus Baukastensystemen wie z.B. Automatisierungselemente wie Ventilinseln (Abbildung 5.1). Produkte mit lauter unterschiedlichen Bauteilen lassen sich ebenso gut abbilden, jedoch erzeugt der Ansatz der Trennung von Beschreibung und Verwendung hier einen Overhead.

Der Aufwand, der für Benutzerinnen und Benutzer mit der Befüllung der Ontologie entsteht richtet sich auch danach, ob Informationen zu den Bauteilen von Produkten aus anderen Datenquellen, wie etwa Bauteilkatalogen oder CAD-Modellen übernommen werden können. Um diese Möglichkeit zu demonstrieren wurde eine Beispiel-Implementierung eines CAD-Importes von Lego®-Modellen vorgestellt. Der Import von Lego®-Modellen hat natürlich nur wenig Praxisrelevanz. Er ist vielmehr als „proof of concept“ und Mittel zu sehen, um umfangreiche Produktstrukturen automatisiert aufzubauen, mit denen die Funktion der Ontologie und der Regeln verifiziert werden kann. Vorteilhaft ist hier vor allem, dass nicht nur Geometrieinformationen in diesen Modellen hinterlegt sind, sondern auch die einzelnen Schritte zum fertigen Modell enthalten sein können und daher auch Zusammenbauschritte automatisiert übernommen werden können. Der Import-Algorithmus verwendet ein fixes Benennungsschema, das die Verifikation der Daten

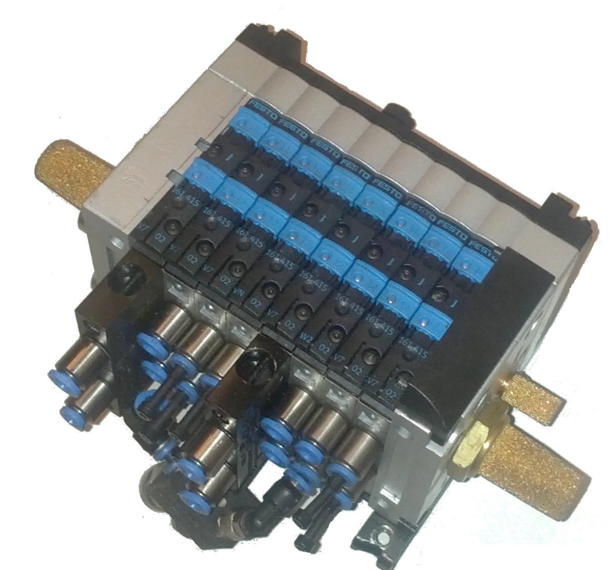


Abbildung 5.1: Ventilinsel (Bild: selbserstellt)

erleichtern soll, etwa indem die Klasse eines Individuums im Namen als Kürzel kodiert ist. Dies führt allerdings zu sehr langen Namen. Obwohl die Verwendung einer regelbasierten Programmierung hier aufgrund der Linearität des Import-Algorithmus nicht von Vorteil ist (und es mit reinem Java-Code wahrscheinlich einfacher lösbar gewesen wäre), wurde der Import-Algorithmus dennoch in Jess umgesetzt. Dadurch lässt er sich wie alle anderen Komponenten des Regelwerks im JessTab benutzen.

Besonderes Augenmerk wurde auf die Ableitung der Planungsschritte für den Zusammenbau gelegt. Dabei war das Ziel sowohl eine sequenzielle Abarbeitung des Zusammenbaus zu unterstützen, als auch Hinweise auf mögliche Parallelisierungen zu geben. Beides konnte mit Hilfe des Regelwerkes umgesetzt werden. Die Verarbeitung ist auch bei komplexeren Produkten mit mehreren tausend Bauteilen in kurzer Zeit abgeschlossen (Prozessor Core i7 2,3Ghz, 16GB RAM).

Das Modell aus Abbildung 5.2 war das größte Modell mit dem die Ontologie und die Regeln getestet wurden. Dies ergab

- insgesamt 3954 Teile, davon 257 verschiedene
- eine Dauer des Imports in Fakten im Arbeitsspeicher von 165 Sekunden
- eine Dauer der Planung der Zusammenbausequenz von 105 Sekunden
- eine Dauer für die Generierung der Bauteilliste von 8 Sekunden

- insgesamt 9865 Individuen, wobei hier von den Geometrieinformationen ausschließlich die Transformationen erzeugt wurden (3606). Für jede `AssemblyDescription` wurde nur ein `AssemblyStep` angelegt, egal wie viele Teile dort gleichzeitig verbaut wurden.
- dass 640 Schritte benötigt werden, um das Produkt zusammenzubauen. Da keine weiteren Informationen zu den Zusammenbauschritten bekannt sind, wird die Dauer für jeden Schritt mit 1 Zeiteinheit festgelegt. Im Worst-Case benötigt der Zusammenbau also 640 Zeiteinheiten.
- einen frühestmöglichen Zeitpunkt der Fertigstellung nach nur 67 Zeiteinheiten, wenn alle Möglichkeiten einer parallelen Verarbeitung genutzt werden (Best-Case).

Bei den Tests stellte sich heraus, dass die Generierung der Individuen problemlos durchgeführt wurde, allerdings für das Speichern der OWL-Datei mit 167800 Zeilen und einer Größe von 11 Megabyte die Standard Stack-Größe von 512kB bei der Ausführung der Java Virtual Machine nicht ausreichte. Für das gezeigte Modell musste sie auf 10 Megabyte erhöht werden. Während der Generierung der Individuen war ein Rechnerkern zu 100% ausgelastet und es wurden insgesamt 1,3 Gigabyte Arbeitsspeicher belegt. Das System lässt sich also auch bei komplexen Produkten mit einem modernen Arbeitsplatzrechner (bei Standard-Arbeitsspeichergrößen von 4 Gigabyte) gut nutzen.

Abbildung 5.2: Komplexeres Testmodell für Ontologie und Regelwerk (Autor: Robert Griebl [51])

5.2 Zusammenfassung

Das Ergebnis dieser Arbeit ist eine Ontologie, die es erlaubt den Aufbau eines mehrteiligen Produktes zu beschreiben. Dieser kann Anwendungen in einer flexiblen Form zur Verfügung gestellt werden. Dabei ist die Ontologie so offen wie möglich gestaltet, um in vielen Einsatzgebieten nützlich zu sein, wie flexiblen Fertigungsanlagen oder der automatisierten Erstellung von Aufbauanleitungen. Es können sowohl Informationen zur Geometrie der Bauteile, als auch Parameter zur Verbindungsart der einzelnen Teile hinterlegt werden. Nach den aufgabenzentrierten Beschreibungen von Teilen der Ontologie in Kapitel 3, kann in Abbildung 5.3 die ganze Ontologie im Gesamtzusammenhang gesehen werden.

Drei wesentliche Fragen zu einem mehrteiligen Produkt werden durch die Ontologie beantwortet:

Wer ist am Zusammenbau beteiligt?

Einzelne Teile können genauso wie ganze Baugruppen in der Ontologie beschrieben werden. Dabei wird eine Trennung vollzogen zwischen der Beschreibung von Teilen und Baugruppen (Klassen `PartDescription` und `AssemblyDescription`), die für jede Art von Bauteil/Baugruppe einzigartig ist. Werden Bauteile bzw. Baugruppen in Produkten verwendet, so wird für jede Verwendung ein `Part`- oder `SubAssembly`-Individuum als Bindeglied zum Produkt oder einer übergeordneten Baugruppenbeschreibung benötigt. Diese Individuen dienen als Anknüpfungspunkt für weitere Informationen zur Integration des Teils bzw. der Baugruppe in das Produkt. Beispielsweise wird dort hinterlegt, wie der Teil im Produkt ausgerichtet sein muss (Abschnitt 3.1ff).

Wie kommt eine Fügung zwischen zwei Bauteilen zu stande?

Jede Beschreibung einer Baugruppe referenziert auf Zusammenbauschritte, die den Zusammenbau der Baugruppe näher beschreiben (Abschnitt 3.2). Dazu kann zu jedem Schritt eine Operation verlinkt werden, wie etwa „Positionieren“, oder „Klebstoff aufbringen“. Hier besteht auch ein Anknüpfungspunkt an Ontologien, die auf einen Zusammenbauprozess spezialisiert sind, wie beispielsweise die Ontologie von Lohse et al. (Abbildung 2.11).

Wann müssen welche Teile in welcher Reihenfolge zusammengefügt werden?

Dieser Frage widmet sich der umfangreichste Teil dieser Arbeit (Abschnitt 3.3), ausgehend von der Motivation einen Zusammenbau für eine flexible Fertigungsanlage zu beschreiben. In der Ontologie bilden die Planungsschritte (`SchedulingStep`s) zu einem Produkt eine Sequenz, in welcher dieses zusammengebaut werden kann. Diese Planungsschritte lassen sich aus den anderen Informationen in der Ontologie ableiten. Zu diesem Zweck wurden Ableitungsregeln entwickelt, die einerseits eine valide Zusammenbausequenz generieren und andererseits Informationen zur Parallelisierbarkeit einzelner Zusammenbauschritte zur Verfügung stellen (Abschnitt 4.2).

Zur Erzeugung von Testdaten zur Verifikation der Implementierungen wurde eine Importfunktion für LDraw-Dateien entwickelt, die exemplarisch zeigt, wie CAD-Daten automatisiert integriert werden können (Abschnitt 4.3).

5.3 Ausblick

Weitere Entwicklungen wären vor allem in Richtung Anwendung der Ontologie interessant, beispielsweise aus den gespeicherten Informationen (semi-)automatisch generierte Steuerungsprogramme für Industrieroboter zu erzeugen. Die vorgestellte Ontologie kann dazu Positions- und Lageinformationen für die einzelnen Teile zur Verfügung stellen. Erweiterungsbedarf besteht aber noch bei der Ermittlung der Verfahrrichtungen, die für den Zusammenbau notwendig sind. Da sich hier recht komplexe Bewegungsmuster z.B. beim Verriegeln eines Bajonett-Verschlusses ergeben können, empfiehlt sich eine Kombination der vorliegenden Ontologie mit entsprechenden bereits existierenden Ontologien zu dieser Thematik wie der von Lohse et al. [9].

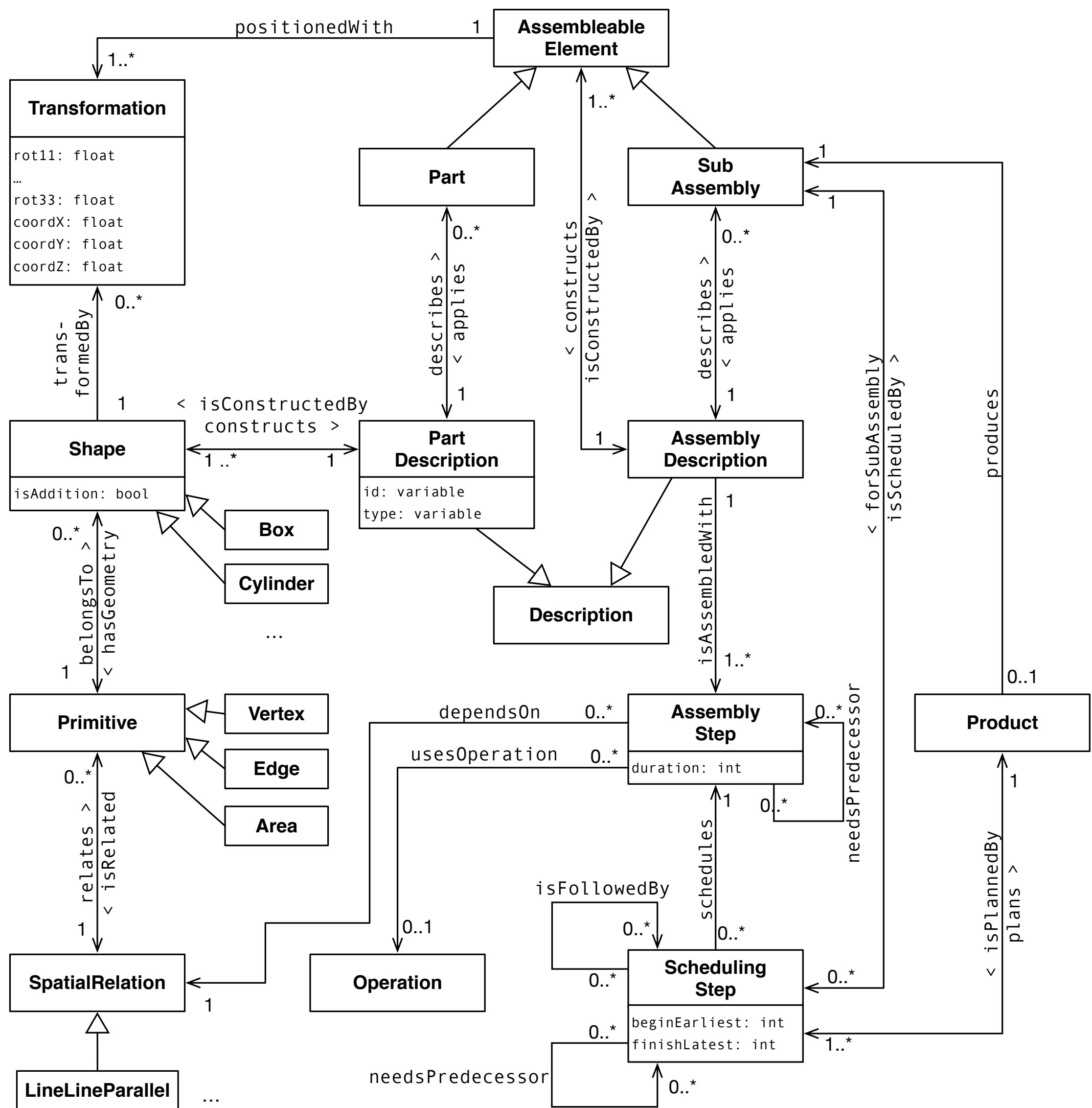


Abbildung 5.3: Gesamte Zusammenbau-Ontologie

Anhang

Programm-Listings

Listings für die Zusammenbauplanung

```
1  (defrule SCHEDULEPRODUCT::createProcessingFactsForProduct
2    ?fldp <- (processedProduct
3      (product ?product)
4      (productname ?productname)
5      (saToProcess $?satoprocess&:(> (length$ $?satoprocess) 0))   ; es gibt noch SAs in
       der Liste
6      (nextSchedulingStepNr ?psn)
7    )
8    =>
9    (bind ?aktsa (nth$ 1 $?satoprocess)) ; zu bearbeitendes SA aus der Liste der noch zu
       verarbeitenden SAs holen
10   (bind $?satoprocess (delete$ $?satoprocess 1 1)) ; und daraus löschen
11   (bind ?aktad (slot-get ?aktsa applies))  ; die damit verknüpfte AD auslesen
12   (bind $?aktas (slot-get ?aktad isAssembledWith)) ; die damit verknüpften AS auslesen
13 ; alle Sub-SAs ermitteln und an die Liste der noch zu verarbeitenden SAs anhängen
14   (bind $?subsa (SCHEDULEPRODUCT::getAllSubSAForSA ?aktsa))
15   (if (> (length$ $?subsa) 0) then
16      (bind $?satoprocess (insert$ $?satoprocess (+ (length$ $?satoprocess) 1) $?subsa))
17   )
18 ; für jeden AS des SA ein Fakt erstellen. Diese Fakts feuern dann die weiteren Regeln
19
20   (foreach ?as $?aktas
21     (assert (processedAssemblyStep
22      (processedProduct ?fldp)
23      (subassembly ?aktsa)
24      (assemblystep ?as)
25     ))
26   )
27 ; veränderte SA-Liste zurückspeichern
28   (modify ?fldp (saToProcess $?satoprocess))
29 )
```

Listing 20: Regel *createProcessingFactsForProduct*

```
1  (defrule SCHEDULEPRODUCT::removeFollowerFromDifferentSAifFollowerFromSameSAexists
2  ; manchmal kann es aufgrund der Regelreihenfolge vorkommen, dass ein SA auf alle SS der
       zugrundeliegenden SAs verweist
3  ; obwohl diese dort gar nicht die "letzten" sind (d.h. es gibt andere SS im selben SA,
       die darauf verweisen)
4  ; diese Regel putzt das aus
5  ; gibt es einen 1. SS
6    (object
7      (is-a SchedulingStep)
8      (OBJECT ?dependantstep)
9      (:NAME ?dependantstepname)
10     (forSubAssembly ?dependantsa)
```

```
11      (isFollowedBy $?dependantfol)
12    )
13  ; und einen 2. SS aus einem ANDEREN SA, der den 1. SS als Vorgänger hat
14    (object
15      (is-a SchedulingStep)
16      (OBJECT ?startstep)
17      (:NAME ?startstepname)
18      (forSubAssembly ?startsa&:(neq ?startsa ?dependantsa))
19      (needsPredecessors $?startpred&:(member$ ?dependantstep $?startpred))
20    )
21  ; und gibt es einen 3. SS aus dem GLEICHEN SA wie der 1. SS, der den 1. SS als Vorgänger
      hat
22    (object
23      (is-a SchedulingStep)
24      (OBJECT ?samesastep)
25      (forSubAssembly ?dependantsa)
26      (needsPredecessors $?samesapred&:(member$ ?dependantstep $?samesapred))
27    )
28    =>
29  ; dann hat der SS aus dem GLEICHEN SA Vorrang und die andere Abhängigkeit wird gelöscht
30    (bind ?posp (member$ ?dependantstep $?startpred))
31    (bind ?posf (member$ ?startstep $?dependantfol))
32    (slot-delete$ ?startstep needsPredecessors ?posp ?posp)
33    (if (neq ?posf FALSE) then
34      (slot-delete$ ?dependantstep isFollowedBy ?posf ?posf)
35    )
36  )
```

Listing 21: Regel *removeFollowerFromDifferentSAifFollowerFromSameSAexists*

```
1   (defrule SCHEDULEPRODUCT::correctFinishLatestForFinalSA
2     (object
3       (is-a SchedulingStep) ; suche einen 1. SS
4       (OBJECT ?minor)
5       (:NAME ?minorname)
6       (finishLatest ?minorlevel&:(neq ?minorlevel nil)) ; mit einem
        Fertigstellungszeitpunkt
7       (forSubAssembly ?sa)
8       (isFollowedBy $?minorfollow&:(eq (length$ $?minorfollow) 0)) ; und keinen Nachfolgern
9     )
10     (object
11      (is-a SchedulingStep) ; such einen 2. SS
12      (OBJECT ?major)
13      (:NAME ?majorname)
14      (finishLatest ?majorlevel&:(and (neq ?majorlevel nil) (> ?majorlevel ?minorlevel)))
        ; dessen Fertigstellungszeitpunkt größer als der des 1. SS ist
15      (forSubAssembly ?sa) ; dem selben SA zugeordnet ist
16      (isFollowedBy $?majorfollow&:(eq (length$ $?majorfollow) 0)) ; und keinen Nachfolger
        hat
17    )
18    =>
19    (slot-set ?minor finishLatest (slot-get ?major finishLatest)) ; alle beendenen SS
        eines SA sollten gleichzeitig aufhören, daher wird der größere Wert kopiert
20  )
```

Listing 22: Regel *correctFinishLatestForFinalSA*

```
1   (defrule SCHEDULEPRODUCT::setFinishLatestLowerThanSuccessor
2     (object
3       (is-a SchedulingStep) ; suche einen SS
4       (OBJECT ?step)
5       (beginEarliest ?elevel&:(neq ?elevel nil)) ; mit einem Beginnzeitpunkt
6       (finishLatest ?llevel&:(neq ?llevel nil)&:(eq ?llevel 0)) ; und einem
        Fertigstellungszeitpunkt 0
7       (:NAME ?stepname)
8       (needsPredecessors $?predecessors)
9       (isFollowedBy $?successor&:(> (length$ $?successor) 0)) ; der Nachfolger hat (also
        nicht die letzten Schritte eines Produktes)
```

```
10    )
11    =>
12    (bind ?newlevel (SCHEDULEPRODUCT::getMinSuccessorLevel ?step)) ; bestimme den
          frühesten Fertigstellungszeitpunkt aller Nachfolger
13    (if (neq ?newlevel nil) then
14  ; sollte sich ein neuer Wert ergeben haben, der größer als 0 ist
15      (if (and (neq ?llevel ?newlevel) (> ?newlevel 0)) then
16        (slot-set ?step finishLatest ?newlevel) ; speichern des neuen
          Fertigstellungszeitpunktes
17  ; allen Vorgängern setzen wir den Fertigstellungszeitpunkt zurück, damit diese neu
          berechnet werden. Dabei dürfen aber die noch nicht fertig angelegten nicht
          verwendet werden
18        (foreach ?predecessor $?predecessors
19          (if (neq (slot-get ?predecessor finishLatest) nil) then
20            (slot-set ?predecessor finishLatest 0)
21          )
22        )
23      )
24    )
25  )
```

Listing 23: Regel *setFinishLatestLowerThanSuccessor*

```
1  (deffunction partlistInsert (?partclass ?partlist)
2    (foreach ?p ?partlist
3      (if (eq (fact-slot-value ?p partname) ?partclass) then
4        (modify ?p (count (+ 1 (fact-slot-value ?p count))))
5        (return ?partlist)
6      )
7    )
8  ; wenn wir hier count gleich auf 1 setzen, dann schlägt u.U. die Fakt-Erstellung fehl
9  ; weil es schon ein Fakt gibt mit diesem Teilnamen und count 1
10 ; daher legen wir es mit 0 an und erhöhen count bei erfolgreichem Anlegen auf 1
11 ; -> es werden für jeden Zählvorgang immer neue Fakten generiert!
12   (bind ?fact (assert (partlistitem (partname ?partclass) (count 0))))
13   (if (neq ?fact FALSE) then
14     (modify ?fact (count (+ 1 (fact-slot-value ?fact count))))
15     (bind ?partlist (insert$ ?partlist 1 ?fact))
16    else
17   )
18   (return ?partlist)
19 )
20
21 (deffunction getPartsOfAssemblyDescription (?assembly ?partlist)
22 ; für alle Descriptions aus denen ein SA aufgebaut ist
23   (foreach ?p (slot-get ?assembly isConstructedBy)
24     (bind ?p (slot-get ?p applies))
25 ; ermittle den Typ des Individuums (seine Klasse)
26     (bind ?partclass (class ?p))
27     (if (eq (isClassOrSubclassOf ?partclass PartDescription) TRUE) then
28 ; ist es eine PD, dann wird die Bauteilliste erweitert
29       (bind ?pname (slot-get ?p :NAME))
30       (bind ?partlist (partlistInsert ?pname ?partlist))
31      else
32 ; sonst ist es ein SA. Für dieses wird die Funktion rekursiv aufgerufen.
33       (bind ?partlist (getPartsOfAssemblyDescription ?p ?partlist))
34     )
35   )
36   (return ?partlist)
37 )
```

Listing 24: Funktion *partlistInsert*

Listings für den LDraw-Import

```
1  (deffunction LDRAWIMPORT::importLDrawAssembly (?filename ?parentassemblyrecord)
2  ; importiert ein SubAssembly aus einer LDraw-Datei.
3  ;    Parameter ?filename: Dateiname der LDraw-Datei
4  ;    Parameter ?parentassemblyrecord: ein bestehendes LDrawAssembly-Template kann als
5  ;       "Parent" übergeben werden. Dies ist dann notwendig, wenn ein LDraw-File auf ein
6  ;       anderes verweist. Im Normalfall kann für den zweiten Paramter FALSE übergeben
       werden.
7    (bind ?an (getBaseNameOfFilepath ?filename))
8    (bind ?pnr 1)
9    (bind ?assemblydata
10     (assert (LDRAWIMPORT::LDrawAssembly
11       (assemblyfilename ?filename)
12       (assemblyname ?an)
13       (filehandle (open ?filename ?an r))
14       (nextSubAssemblyNr 1)
15       (nextAssemblyDescriptionNr 1)
16       (nextAssemblyStepNr 1)
17       (nextPartNr ?pnr)
18       (activeAssemblyDescription nil)
19       (activeAssemblyStep nil)
20     (activeFileLine (list))
21       (createNewAssemblyStep TRUE)
22       (createNewPart FALSE)
23       (addExistingAssembly FALSE)
24       (parentAssembly ?parentassemblyrecord)
25       )
26     )
27   )
28   (focus LDRAWIMPORT)
29   (run)
30   (focus MAIN)
31   (return ?assemblydata)
32 )
```

Listing 25: Funktion *importLDrawAssembly*

```
1  (deffunction LDRAWIMPORT::getTransformationMatrix ($?fileline)
2  ; liest eine homogene Transformationsmatrix aus einer Referenzzeile und gibt diese
3  ; als Transformation-Individual zurück.
4  ; als Sonderfall wird die Identitäts-Matrix (keine Transformation) zurückgegeben, wenn
       der Parameter NIL oder eine leere Liste ist
5
6    (if (and (neq $?fileline nil) (neq (length$ $?fileline) 0)) then
7    (bind ?coordx (float (nth$ 3 $?fileline)))
8    (bind ?coordy (float (nth$ 4 $?fileline)))
9      (bind ?coordz (float (nth$ 5 $?fileline)))
10     (bind ?rot11 (float (nth$ 6 $?fileline)))
11     (bind ?rot12 (float (nth$ 7 $?fileline)))
12     (bind ?rot13 (float (nth$ 8 $?fileline)))
13     (bind ?rot21 (float (nth$ 9 $?fileline)))
14     (bind ?rot22 (float (nth$ 10 $?fileline)))
15     (bind ?rot23 (float (nth$ 11 $?fileline)))
16     (bind ?rot31 (float (nth$ 12 $?fileline)))
17     (bind ?rot32 (float (nth$ 13 $?fileline)))
18     (bind ?rot33 (float (nth$ 14 $?fileline)))
19     (bind ?name (str-cat ?*PrefixTransformation* ?coordx "_" ?coordy "_" ?coordz "_" ?
       rot11 "_" ?rot12 "_" ?rot13 "_" ?rot21 "_" ?rot22 "_" ?rot23 "_" ?rot31 "_" ?rot32
       "_" ?rot33))
20     (bind ?m (nth$ 1 (find-instance ((?h Transformation)) (eq (slot-get ?h :NAME) ?
       name))))
21     (if (eq ?m nil) then
22       (bind ?m (make-instance ?name of Transformation
23         (coordX ?coordx) (coordY ?coordy) (coordZ ?coordz)
24         (rot11 ?rot11) (rot12 ?rot12) (rot13 ?rot13)
25         (rot21 ?rot21) (rot22 ?rot22) (rot23 ?rot23)
26         (rot31 ?rot31) (rot32 ?rot32) (rot33 ?rot33)
27       ))
28     )
```

```
29      else
30        (bind ?name (str-cat ?*PrefixTransformation* "NoTransformation"))
31        (bind ?m (nth$ 1 (find-instance ((?h Transformation)) (eq (slot-get ?h :NAME) ?
          name))))
32        (if (eq ?m nil) then
33          (bind ?m (make-instance ?name of Transformation
34            (coordX 0.0) (coordY 0.0) (coordZ 0.0)
35            (rot11 1.0) (rot12 0.0) (rot13 0.0)
36            (rot21 0.0) (rot22 1.0) (rot23 0.0)
37            (rot31 0.0) (rot32 0.0) (rot33 1.0)
38          ))
39        )
40      )
41      (return ?m)
42  )
```

Listing 26: Funktion *getTransformationMatrix*

```
1  (deffunction LDRAWIMPORT::createNewAssemblyStep (?ldrawimport)
2  ; erstellt einen neuen AssemblyStep und fügt ihn beim aktiven Assembly dazu
3  ; der neue AssemblyStep wird außerdem gleich aktiv gesetzt
4    (bind ?aa (fact-slot-value ?ldrawimport activeAssemblyDescription))
5    (bind ?aas (fact-slot-value ?ldrawimport activeAssemblyStep))
6    (bind ?asnr (fact-slot-value ?ldrawimport nextAssemblyStepNr))
7    (bind ?an (fact-slot-value ?ldrawimport assemblyname))
8    (bind ?asname (str-cat ?*PrefixAssemblyStep* ?an "_" ?asnr))
9    (bind ?asnew (make-instance ?asname of AssemblyStep (needsPredecessors ?aas) (builds ?
       aa)))
10   (modify ?ldrawimport
11     (activeAssemblyStep ?asnew)
12     (nextAssemblyStepNr (+ ?asnr 1))
13   )
14   (return ?asnew)
15 )
```

Listing 27: Funktion *createNewAssemblyStep*

```
1  (defrule LDRAWIMPORT::readFromFile
2  ; liest eine neue Zeile aus der LDraw-Datei, sobald die aktuelle "verbraucht"
3  ; wurde d.h. sobald alle Teile der aktuellen Zeile abgearbeitet wurden
4    ; nur dann eine neue Zeile aus der Datei lesen, wenn
5    ?fldr <- (LDrawAssembly
6              (assemblyfilename ?an)
7              (filehandle ?fhandle)
8                  (activeFileLine $?line
9    ; - noch nie eine gelesen wurde (Dateianfang)
10              &:(neq $?line nil)
11   ; - andere Regeln mit dem abarbeiten der aktiven Zeile fertig sind (Länge 0)
12              &:(eq 0 (length$ $?line))
13                )
14              )
15   =>
16   ; die neue Zeile aus der Datei lesen und auf die Factbase legen
17   (modify ?fldr (activeFileLine (explode$ (readline ?fhandle))))
18 )
```

Listing 28: Regel *readFromFile*

```
1  (defrule LDRAWIMPORT::commentOrMetaLine
2  ; wertet Kommentar- bzw. Metacode-Zeilen in LDRaw-Dateien aus
3  ; Diese Zeilen entscheiden z.B. wann ein neuer SubAssembly-Schritt beginnt
4    ; wir haben eine Metacode/Kommentar-Zeile wenn
5    ?fldr <- (LDrawAssembly
6      (assemblyfilename ?afn)
7      (activeFileLine $?line
8        &:(neq $?line nil)
9      ; an erster Stelle Code "0" steht
```

```
10        &:(eq 0 (nth$ 1 $?line))
11      )
12      (createNewAssemblyStep FALSE)
13    )
14    =>
15    ; Information zu Assemblysteps sind durch das Schlüsselwort "STEP" gekennzeichnet
16    (if (str-index "STEP" (implode$ (rest$ $?line))) then
17      ; Trigger-Fact für newAssemblyStep-Rule setzen
18    (modify ?fldr
19        (createNewAssemblyStep TRUE)
20      )
21     else
22      ; restliche aktive Zeile leeren, damit nachste Zeile eingelesen wird
23      (modify ?fldr (activeFileLine (list)))
24    )
25  )
```

Listing 29: Regel *commentOrMetaLine*

```
1  (defrule LDRAWIMPORT::createNewPart
2  ; erstellt mit den Informationen aus der aktiven Zeile einen neuen Part.
3  ; die Part-Klasse muss in der Ontology existieren (wird über Namen gematcht)
4    ?fldr <- (LDrawAssembly
5      (createNewPart ?partname&:(neq ?partname FALSE))
6      (createNewAssemblyStep FALSE)
7      (activeAssemblyDescription ?aa)
8      (activeAssemblyStep ?aas)
9      (nextPartNr ?pnr)
10     (activeFileLine $?afl)
11     (assemblyname ?an)
12     (transformation ?trans)
13   )
14   =>
15   ; ist der referenzierte Teil auch wirklich ein Part und ist die PartDescription schon
       in der Ontologie vorhanden?
16   (bind ?partname2 (str-cat ?*PrefixPartDescription* ?partname))
17   (if (not (instance-existp ?partname2)) then
18     (make-instance ?partname2 of PartDescription map)
19   )
20   (bind ?pd (instance-address ?partname2))
21   (bind ?pn (str-cat ?*PrefixPart* ?an "-" ?pnr "_" ?partname))
22
23     ; hat der Teil eine Positionierungsinformation (eigentlich immer), dann müssen wir
       ein Positioning-Element erstellen
24   (bind ?p (make-instance ?pn of Part (positionedWith ?trans) (applies ?pd)))
25     ; jetzt noch den Teil zur aktiven AssemblyDescription hinzufügen
26   (slot-insert$ ?aa isConstructedBy 1 ?p)
27
28   ; PartDescription gibt es noch nicht, also wird sie angelgt. Um die Geometrie wird
       sich später gekümmert (GEOMETRY)
29
30
31   ; Teilenummer erhöhen und createNewPart zurücksetzen
32   ; restliche aktive Zeile leeren, damit nächte Zeile eingelesen wird
33   (modify ?fldr
34     (nextPartNr (+ ?pnr 1))
35     (createNewPart FALSE)
36     (activeFileLine (list))
37     (transformation nil)
38   )
39 )
```

Listing 30: Regel *createNewPart*

```
1  (defrule LDRAWIMPORT::addExistingAssembly
2  ; der Name der Regel ist nicht ganz korrekt, da falls ein SubAssembly noch nicht
3  ; existiert dieses importiert wird.
4    ?fldr <- (LDrawAssembly
```

```
5      (addExistingAssembly ?assemblyname&:(neq ?assemblyname FALSE))
6      (activeAssemblyDescription ?aa)
7      (activeAssemblyStep ?aas)
8      (transformation ?trans)
9      (nextAssemblyStepNr ?asnr)
10     (nextSubAssemblyNr ?sanr)
11     (assemblyname ?an)
12   )
13   =>
14   ; sicherheitshalber eliminieren wir hier nochmals den Pfad, falls in der
       Parent-LDraw-Datei eine Pfad angegeben wurde.
15   (bind ?assemblyfile (str-cat "./models/" ?assemblyname))
16   (bind ?assemblyname (getBaseNameOfFilepath ?assemblyfile))
17   (bind ?adn (str-cat ?*PrefixAssemblyDescription* ?assemblyname))
18   ; zuerst nachsehen, ob AssemblyDescription überhaupt aus einer Datei importiert werden
       muss. Ev ist sie ja schon in
19   ; der Ontologie enthalten (Nur Namensvergleich!)
20   (if (instance-existp ?adn) then
21     (bind ?assemblyinstance (instance-address ?adn))
22     (bind ?san (str-cat "SA-" ?an "-" ?sanr "_" ?adn))
23     (bind ?sa (make-instance ?san of SubAssembly (positionedWith ?trans) (applies ?
       assemblyinstance) ))
24     (slot-insert$ ?aa isConstructedBy 1 ?sa)
25     (modify ?fldr
26       (nextSubAssemblyNr (+ ?sanr 1))
27       (addExistingAssembly FALSE)
28       (activeFileLine (list))
29       (transformation nil)
30     )
31    else
32     ; die Verbindung der Assemblies wird im Subassembly gesetzt, daher ist hier weiter
       nichts zu tun, als
33     ; die Erstellung des Subassemblies aufzurufen.
34     ; Das Parent-Element darf solange NICHT weiterbearbeitet werden!
35     ; ein SubAssembly für diese neu importierte AssemblyDescription wird beim
       Finalisieren gesetzt, muss also nicht hier explizit angegeben werden.
36 ; TODO hardgecodeten Pfad hier raus und durch sinnvolleres ersetzen
37     (bind ?newfldr (LDRAWIMPORT::importLDrawAssembly ?assemblyfile ?fldr))
38   )
39 )
```

Listing 31: Regel *addExistingAssembly*

```
1 (defrule LDRAWIMPORT::createNewAssemblyDescriptionAndAssemblyStep
2 ; erstellt neue Assemblies und Assemblysteps sobald eine Zeile in der LDraw-Datei
       anzeigt, dass
3 ; ein neuer Bearbeitungsschritt gemacht werden soll.
4   ?fldr <- (LDrawAssembly
5     (createNewAssemblyStep TRUE)
6     (activeAssemblyDescription ?aold)
7     (activeAssemblyStep ?asold)
8     (nextSubAssemblyNr ?sanr)
9     (nextAssemblyDescriptionNr ?adnr)
10    (nextAssemblyStepNr ?asnr)
11    (nextSubAssemblyNr ?sanr)
12    (activeFileLine $?afl)
13    (assemblyname ?an)
14  )
15  =>
16  ; Individual für die AssemblyDescription erstellen
17  (bind ?adn (str-cat ?*PrefixAssemblyDescription* ?an "_sub" ?adnr))
18  (bind ?anew (make-instance ?adn of AssemblyDescription))
19  ; Individual für den AssemblyStep erstellen
20  (bind ?name (str-cat ?*PrefixAssemblyStep* ?an "_" ?adnr))
21  (bind ?asnew (make-instance ?name of AssemblyStep))
22  ; gab es schon vorher eine AssemblyDescription?
23  (if (and (neq ?aold nil) (neq ?asold nil)) then
24 ;     (slot-set ?aold endsWith ?asold)
25    ; Individual für das SubAssembly erstellen
```

```
26       (bind ?name (str-cat "SA-" ?an "-" ?sanr "_" (slot-get ?aold :NAME)))
27       (bind ?sanew (make-instance ?name of SubAssembly (applies ?aold)))
28       (slot-insert$ ?anew isConstructedBy 1 ?sanew)
29       (modify ?fldr
30         (nextSubAssemblyNr (+ ?sanr 1))
31       )
32     else
33   )
34   ; (slot-set ?anew startsWith ?asnew)
35   (slot-set ?asnew builds ?anew)
36   (modify ?fldr
37       (activeAssemblyDescription ?anew)
38       (activeAssemblyStep ?asnew)
39       (nextAssemblyDescriptionNr (+ ?adnr 1))
40       (nextAssemblyStepNr (+ ?asnr 1))
41       (createNewAssemblyStep FALSE)
42       ; restliche aktive Zeile leeren, damit nächste Zeile eingelesen wird
43   (activeFileLine (list))
44   )
45 )
```

Listing 32: Regel *createNewAssemblyDescriptionAndAssemblyStep*

```
1  (defrule LDRAWIMPORT::finishAssembly
2  ; wird aufgerufen, wenn die Datei fertig gelesen ist.
3  ; setzt letzte Slots (isReadyWith, ..) für das Produkt
4    ?fldr <- (LDrawAssembly
5    (activeFileLine ?line
6          &:(eq ?line EOF)
7      )
8      (activeAssemblyDescription ?aa)
9      (nextSubAssemblyNr ?sanr)
10     (assemblyname ?an)
11     (activeAssemblyStep ?aas)
12     (filehandle ?fhandle)
13     (parentAssembly ?parent)
14     (transformation ?trans)
15   )
16   =>
17   ; Das letzte SubAssembly steht für die ganze importierte Datei -> die Nummer wird aus
       dem Namen entfernt.
18   (bind ?realname (str-cat ?*PrefixAssemblyDescription* ?an))
19   (slot-set ?aa :NAME ?realname)
20 ;  (slot-set ?aa endsWith ?aas)
21   (if (neq ?parent FALSE) then
22     ; wir bearbeiten gerade ein Subassembly, d.h. einen Dateiimport, der von einer
         anderen Datei ausgelöst wurde.
23
24     (bind ?paa (fact-slot-value ?parent activeAssemblyDescription))
25     (bind ?pan (fact-slot-value ?parent assemblyname))
26     (bind ?psanr (fact-slot-value ?parent nextSubAssemblyNr))
27     ; Transformations-Matrix aus der aktiven Zeile des PARENTS lesen und in Ontologie
         ablegen
28     (bind ?trans (fact-slot-value ?parent transformation))
29     (bind ?an (str-cat "SA-" ?pan "-" ?psanr "_" (slot-get ?aa :NAME)))
30     (bind ?sa (make-instance ?an of SubAssembly (positionedWith ?trans) (applies ?aa)))
31     (slot-insert$ ?paa isConstructedBy 1 ?sa)
32
33     ; jetzt müssen wir noch die Trigger zurücksetzen, damit das Haupt-SubAssembly
         weiterarbeiten kann.
34     (modify ?parent
35       (nextSubAssemblyNr (+ ?psanr 1))
36       (addExistingAssembly FALSE)
37       (activeFileLine (list))
38       (transformation nil)
39     )
40   )
41   ; Aufräumarbeiten
42   (close ?fhandle)
43   (retract ?fldr)
```

```
44 )
```

Listing 33: Regel *finishAssembly*

```
1  (defrule LDRAWIMPORT::setIdentTransformation
2  ; setzt für alle Elemente ohne Transformationsinformationen die identische Transformation
3  ; damit einheitlich zugegriffen werden kann.
4    (object
5      (is-a ?c &:(isClassOrSubclassOf ?c AssembleableElement))
6      (positionedWith ?t &:(eq ?t nil))
7      (OBJECT ?ae)
8      (:NAME ?aename)
9    )
10   =>
11   (slot-set ?ae positionedWith (LDRAWIMPORT::getTransformationMatrix (create$)))
12 )
```

Listing 34: Regel *setIdentTransformation*

Listings für die Geometrieableitungen

```
1  (deffunction GEOMETRY::createPrimitivesForAssembleableElement (?name)
2  ; erzeugt alle einfachen Primitive (Linien, Kanten, Ebenen, Flächen) für das angegebene
       AE
3  ; es wird nur für diese Ebene erzeugt inklusive aller Transformationen eingerechnet.
4  ; die Primitive für alle Geometrien zu erzeugen wäre zu langwierig
5  ; ACHTUNG: die Primitive werden nicht aktualisiert, sollte sich am zugrundeliegenden
       Assemblybaum etwas ändern
6    (bind ?assembleable (nth$ 1 (find-instance ((?h AssembleableElement)) (eq (slot-get ?h
       :NAME) ?name))))
7    (focus GEOMETRY)
8    (if (neq ?assembleable nil) then
9      (assert (GEOMETRY::PrimitiveContainer
10       (owner ?assembleable)
11       (activeAE ?assembleable)
12       (primitives (create$))
13       (activetransformation (JAVA3D::convPTransformation_JTransform3D (slot-get ?
       assembleable positionedWith)))
14     ))
15
16     (run)
17   else
18   )
19   (focus MAIN)
20 )
```

Listing 35: Funktion *createPrimitivesForAssembleableElement*

```
1  (deffunction GEOMETRY::assignVertex (?x ?y ?z ?owner)
2  ; kontrolliert, ob ein Punkt in der Ontologie schon vorhanden ist
3  ; wenn ja, wird dieser zurückgeliefert, wenn nicht, dann wird er erzeugt
4    (bind ?name (str-cat "V-" ?x "-" ?y "-" ?z "_" (slot-get ?owner :NAME)))
5    (bind ?vertex (nth$ 1 (find-instance ((?v Vertex))
6      (or (eq (slot-get ?v :NAME) ?name)
7      (and (eq (slot-get ?v coordX) ?x)
8        (and (eq (slot-get ?v coordY) ?y)
9          (and (eq (slot-get ?v coordZ) ?z)
10           (eq (slot-get ?v belongsTo) ?owner)
11         )
12       )
13     ))
14   )))
15   (if (eq ?vertex nil) then
16     ; nichts gefunden -> neuen Vertex erstellen.
17      (bind ?vertex (make-instance ?name of Vertex (coordX ?x) (coordY ?y) (coordZ ?z)
       (belongsTo ?owner)))
18   else
19   )
20   (return ?vertex)
21 )
```

Listing 36: Funktion *assignVertex*

```
1  (defrule GEOMETRY::generatePrimitivesBox
2  ; erzeugt alle Primitive eines Quaders (8 Vertices, 12 Edges, 6 Areas)
3  ; mit allen Abhängigkeiten.
4    (object
5      (is-a Box)
6      (OBJECT ?obj)
7      (:NAME ?name)
8      (transformedBy ?trans &:(neq ?trans nil))
9      (dimensionX ?dx)
10     (dimensionY ?dy)
11     (dimensionZ ?dz)
12     (hasGeometry $?c &:(eq (length$ $?c) 0))
13   )
14   =>
```

```
15  ; Generiere Eckpunkte
16    (bind ?v0 (GEOMETRY::assignVertex 0.0 0.0 0.0 ?obj))
17    (bind ?vx (GEOMETRY::assignVertex ?dx 0.0 0.0 ?obj))
18    (bind ?vy (GEOMETRY::assignVertex 0.0 ?dy 0.0 ?obj))
19    (bind ?vz (GEOMETRY::assignVertex 0.0 0.0 ?dz ?obj))
20    (bind ?vxy (GEOMETRY::assignVertex ?dx ?dy 0.0 ?obj))
21    (bind ?vxz (GEOMETRY::assignVertex ?dx 0.0 ?dz ?obj))
22    (bind ?vyz (GEOMETRY::assignVertex 0.0 ?dy ?dz ?obj))
23    (bind ?vxyz (GEOMETRY::assignVertex ?dx ?dy ?dz ?obj))
24    (JAVA3D::PTransformPVertices ?trans (create$ ?v0 ?vx ?vy ?vz ?vxy ?vxz ?vyz ?vxyz))
25  ; Generiere Kanten
26    (bind ?e0_x (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?v0 ?vx))))
27    (bind ?e0_y (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?v0 ?vy))))
28    (bind ?e0_z (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?v0 ?vz))))
29    (bind ?ex_xy (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vx ?vxy))))
30    (bind ?ex_xz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vx ?vxz))))
31    (bind ?ey_yz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vy ?vyz))))
32    (bind ?ey_xy (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vy ?vxy))))
33    (bind ?ez_xz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vz ?vxz))))
34    (bind ?ez_yz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vz ?vyz))))
35    (bind ?exy_xyz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vxy ?
        vxyz))))
36    (bind ?exz_xyz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vxz ?
        vxyz))))
37    (bind ?eyz_xyz (make-instance of Edge (belongsTo ?obj) (hasVertices (create$ ?vyz ?
        vxyz))))
38  ; Generiere Flächen
39    (bind ?p1 (make-instance of Area (belongsTo ?obj) (hasEdges (create$ ?e0_x ?ex_xz ?
        ez_xz ?e0_z)) (hasVertices (create$ ?v0 ?vx ?vxz ?vz))))
40    (bind ?p2 (make-instance of Area (belongsTo ?obj) (hasEdges (create$ ?e0_x ?ex_xy ?
        ey_xy ?e0_y)) (hasVertices (create$ ?v0 ?vx ?vxy ?vy))))
41    (bind ?p3 (make-instance of Area (belongsTo ?obj) (hasEdges (create$ ?e0_z ?ez_yz ?
        ey_yz ?e0_y)) (hasVertices (create$ ?v0 ?vz ?vyz ?vy))))
42    (bind ?p4 (make-instance of Area (belongsTo ?obj) (hasEdges (create$ ?ex_xy ?exy_xyz ?
        exz_xyz ?ex_xz)) (hasVertices (create$ ?vx ?vxy ?vxyz ?vxz))))
43    (bind ?p5 (make-instance of Area (belongsTo ?obj) (hasEdges (create$ ?ey_xy ?exy_xyz ?
        eyz_xyz ?ey_yz)) (hasVertices (create$ ?vy ?vxy ?vxyz ?vyz))))
44    (bind ?p6 (make-instance of Area (belongsTo ?obj) (hasEdges (create$ ?ez_xz ?exz_xyz ?
        eyz_xyz ?ez_yz)) (hasVertices (create$ ?vz ?vxz ?vxyz ?vyz))))
45  )
```

Listing 37: Regel *generatePrimitivesBox*

```
1  (defrule GEOMETRY::joinVertices
2    (object
3      (is-a Vertex)
4      (OBJECT ?v1)
5      (:NAME ?v1name)
6      (belongsTo ?owner)
7      (coordX ?x)
8      (coordY ?y)
9      (coordZ ?z)
10   )
11   (object
12     (is-a Vertex)
13     (OBJECT ?v2 &: (neq ?v1 ?v2))
14     (:NAME ?v2name)
15     (belongsTo ?owner)
16     (coordX ?x)
17     (coordY ?y)
18     (coordZ ?z)
19   )
20   (object
21     (is-a ?t1 &:(isClassOrSubclassOf ?t1 Primitive))
22     (OBJECT ?primitive)
23     (belongsTo ?owner)
24     (hasVertices $?vertices &: (neq (member$ ?v2 $?vertices) FALSE))
25   )
26   =>
27   (bind ?v2pos (member$ ?v2 $?vertices))
28   (delete$ ?v2pos ?v2pos $?vertices)
29   (if (eq (member$ ?v1 $?vertices) FALSE) then
```

```
30      (insert$ ?v1 $?vertices)
31    )
32    (slot-set ?primitive hasVertices $?vertices)
33  )
```

Listing 38: Regel *joinVertices*

```
1  (deffunction JAVA3D::JTransformPrimitivesToDuplicate (?owner ?
       javatransform3d $?primitives)
2  ; dupliziert mehrere Primitive (Protege), transformiert diese mittels einer
       Transformation (Java3D)
3  ; und hängt sie an ein neues AssembleableElement oder Shape (?owner) an
4  ; schreibt die transformierten Koordinaten in den Vektor zurück.
5  ; liefert die duplizierten und transformierten Primitive zurück
6    (bind ?duplicates (create$))
7    (if (and (neq $?primitives nil) (neq (length$ $?primitives) 0)) then
8      (foreach ?primitive $?primitives
9        ; primitive duplizieren
10       (bind ?newprimitive (duplicate-instance (slot-get ?primitive :NAME) (belongsTo ?
      owner)))
11       (bind $?vertices (slot-get ?newprimitive hasVertices))
12       ; alte Vertices löschen
13       (slot-set ?newprimitive hasVertices (create$))
14       ; jetzt noch alle enthaltenen Vertices duplizieren.
15       (foreach ?vertex $?vertices
16         (bind ?newvertex (GEOMETRY::assignVertex (slot-get ?vertex coordX) (slot-get ?
      vertex coordY) (slot-get ?vertex coordZ) ?owner))
17         (slot-insert$ ?newprimitive hasVertices 1 ?newvertex)
18       )
19       (JAVA3D::JTransformPVertices ?javatransform3d (slot-get ?newprimitive hasVertices))
20       (bind ?duplicates (insert$ ?duplicates 1 ?newprimitive))
21     )
22   else
23   )
24   (return ?duplicates)
25 )
```

Listing 39: Funktion *JTransformPrimitivesToDuplicate*

```
1  (deffunction JAVA3D::JTransformPVertices (?javatransform3d $?vertices)
2  ; transformiert mehrere Vektoren (Protege) mittels einer Transformation (Java3D)
3  ; schreibt die transformierten Koordinaten in die Vektoren zurück.
4  ; liefert nichts zurück
5    (foreach ?vector $?vertices
6      (bind ?vec (JAVA3D::convPVertex_JPoint3f ?vector))
7      (call ?javatransform3d transform ?vec)
8      (slot-set ?vector coordX (get-member ?vec x))
9      (slot-set ?vector coordY (get-member ?vec y))
10     (slot-set ?vector coordZ (get-member ?vec z))
11   )
12 )
```

Listing 40: Regel *JTransformPVertices*

Abbildungsverzeichnis

1.1 Beispiele für modulare Produkte 2

1.2 Positionierung eines Bolzens in einem Lagerbock 3

2.1 „Semantische Treppe“ [12] 10

2.2 Beispiel einer Taxonomie [13] 11

2.3 Beispiel eines Thesaurus [13] 11

2.4 Beispiel einer Topic Map [13] 12

2.5 Schichten des Semantic Web nach Tim Berners-Lee [13] 13

2.6 Beispiel einer Ontologie [13] 13

2.7 Teil der Ontologie nach [7] zur Beschreibung eines Produktes 17

2.8 Räumliche Abhängigkeiten („Spatial Relationships“) und deren Verbindungsmerkmale („Mating Features“) nach [26] 18

2.9 Klassen der AsD-Ontologie nach [27] 19

2.10 Beziehungen zwischen Komponenten mit „Ports“ nach [28] 20

2.11 Ausschnitte aus der Ontologie für die Beschreibung von Zusammenbauprozessen nach [9] 21

2.12 Zusammenbauanleitung eines TV-Tisches [30] 22

3.1 Darstellung von Klassen und Beziehungen zwischen Klassen in UML-Notation 24

3.2 Produktbeschreibung anhand einer (unvollständigen) Bauteilliste 25

3.3 Produktbeschreibung anhand von funktionalen Baugruppen 26

3.4 Logische Bauteil-Hierarchie 26

3.5 Ausschnitt aus der Ontologie: `AssembleableElement` und `Description` 27

3.6 PKW mit vier Rädern: Zusammenhang zwischen `PartDescription` (PD), `Part` (P), `AssemblyDescription` (AD) und `SubAssembly` (SA) 30

3.7 Alternative Version des PKW mit vier Rädern: alle Räder werden in einem Schritt montiert 31

3.8 Jedes `AssembleableElement` ist über eine Transformationsmatrix geometrisch in einer Baugruppe verortet 31

3.9 `AssemblyStep` bilden die Information für Zusammenbauschritte ab 32

3.10 Fertigungsoperationen in einer Ontologie für flexibel Fertigungsanlagen (Bildausschnitt [32]) . . . 33
3.11 Beispiel einer `AssemblyDescription` (AD) mit zugeordneten `AssemblyStep`s (AS) . . . 34
3.12 Varianten für die Verkettung von `AssemblyStep`s (Grafiken des Tisches aus [30]) 35
3.13 Ausschnitt aus der Ontologie mit den Klassen zur Planung des Zusammenbaus 36
3.14 Montage des ersten Rades des Fahrzeuges mit zugehörigen `AssemblyStep`s (AS) und daraus generierten `SchedulingStep`s (SS). . . . 38
3.15 Geometrieklassen und deren Beziehungen . . . 44
3.16 Geometrie eines klassischen „Vierer-Legosteins“ . . . 45

4.1 OWL-Klassenansicht in Protégé . . . 49
4.2 Jambalaya-Ansicht in Protégé mit mehreren Individuen und Beziehungen zwischen ihnen. . . . 50
4.3 JessTab - Integration von Jess in Protégé . . . 51
4.4 JessDE - eine Jess-Integration in Eclipse . . . 52
4.5 Unterschiedliche Reihenfolgen beim Feuern der Regeln kann zu falschen Ergebnissen führen. . . . 56
4.6 Geometrische Komponenten eines 1x2-Legosteins im Flächenmodell des LDraw-Formats (Bild: Holly-Wood.it [50]) . . . 61
4.7 Test-Modell aus Listing 16. Der rote Baustein ist gegenüber dem darunterliegenden blauen um 30 Grad gedreht. . . . 62
4.8 Bauteilhierarchie eines Produktes und die dazu erzeugte Bauteilliste . . . 66
4.9 Java-Applikation zur einfachen Handhabung der Jess-Regelbasis für die Zusammenbau-Ontologie . . . 67

5.1 Ventilinsel (Bild: selbserstellt) . . . 70
5.2 Komplexeres Testmodell für Ontologie und Regelwerk (Autor: Robert Griebl [51]) . . . 71
5.3 Gesamte Zusammenbau-Ontologie . . . 74

Akronyme

CWA	Closed World Assumption
Jess	Jess®, the Rule Engine for the Java Platform
NIST	National Institute of Standards and Technology
OKBC	Open Knowledge Base Connectivity
OWA	Open World Assumption
OWL	OWL Web Ontology Language
PSL	Process Specification Language
RDF	Ressource Description Framework
RDFS	RDF Schema
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
SWRL	Semantic Web Rule Language
Tag	Schlagwort
UML	Unified Modelling Language
URI	Uniform Ressource Identifier
WWW	World Wide Web
XML	eXtensible Markup Language

Literaturverzeichnis

[1] E. Uhlmann, “Wandel der Fabrik durch Produkt-individualisierung,” in *Marktchance Individualisierung*, G. Reinhart and M. Zäh, Eds. Springer Berlin Heidelberg, 2003, pp. 119–127. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-55495-7_11

[2] W. Kersten and E.-M. Kern, “Flexibilität in der verteilten Produktentwicklung,” in *Erfolgsfaktor Flexibilität.* Erich Schmidt Verlag, 2005, pp. 229–250.

[3] IKEA Möbelvertrieb OHG. Ikea PAX-Planer. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.ikea.com/ms/de_AT/rooms_ideas/planner_pax3d/index.html

[4] G. Pahl, *Konstruieren mit 3D-CAD-Systemen: Grundlagen, Arbeitstechnik, Anwendungen.* Springer-Verlag, 1990.

[5] R. D. Sriram and S. Szykman, “Design Repositories and Product Representation for Collaborative Product Development,” in *Proceedings of the Sixth International Conference on Computer Supported Cooperative Work in Design*, W. Shen, Z. Lin, J.-P. Barthès, and M. Kamel, Eds., 2001.

[6] M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents.* John Wiley and Sons Ltd, 2005.

[7] J. Lin, M. Fox, and T. Bilgic, “A Requirement Ontology for Engineering Design,” *Concurrent Engineering Research and Applications*, vol. 4, no. 3, pp. 279–291, 1996.

[8] “ÖNORM EN ISO 10303-210: 2003 04 01, Industrial automation systems and integration - Product data representation and exchange - Part 210: Application protocol: Electronic assembly, interconnection, and packaging design (ISO 10303-210:2001, not included).”

[9] N. Lohse, H. Hirani, S. Ratchev, and M. Turitto, “An Ontology for the Definition and Validation of Assembly Processes for Evolvable Assembly Systems,” University of Nottingham, United Kingdom, Tech. Rep., 2005.

[10] O. Ganschar, S. Gerlan, M. Hämmerle, T. Krause, and S. Schlund, *Produktionsarbeit der Zukunft - Industrie 4.0*, D. Spath, Ed. Fraunhofer Verlag, 2013.

[11] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web: a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities." *Scientific American*, vol. 284, pp. S. 34–43, Mai 2001.

[12] A. Blumauer and T. Pellegrini, *Semantic Web und semantische Technologien. Zentrale Begriffe und Unterscheidungen.* Berlin: Springer Verlag, 2006, pp. 9 – 27.

[13] M. Ullrich, A. Maie, and J. Angele, "Taxonomie, Thesaurus, Topic Map, Ontologie – ein Vergleich v1.4," ontoprise® GmbH (www.ontoprise.de), Tech. Rep., 2004.

[14] (2010) Topic Maps. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.topicmaps.org

[15] (2006) ISO/IEC 13250-2:2006 - Topic Maps - Part 2: Data model. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.isotopicmaps.org/sam/

[16] T. R. Gruber. (1993) What is an Ontology? Zuletzt besucht: 30.10.2015. [Online]. Available: http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[17] W3C: World Wide Web Consortium. (2014) RDF 1.1 Semantics. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.w3.org/TR/rdf11-mt/

[18] ——. (2014) RDF Schema 1.1. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.w3.org/TR/rdf-schema/

[19] ——. (2012, 12) OWL 2 Web Ontology Language. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.w3.org/TR/owl2-overview/

[20] N. D. N. and R. Shearer. (2006) The Open World Assumption. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.nesc.ac.uk/talks/701/OWA_NDrummond.pdf

[21] N. Noy and D. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology," Knowledge Systems Laboratory, Stanford University, Tech. Rep., 3 2001.

[22] M. Gruninger and M. Fox, "Methodology for the Design and Evaluation of Ontologies," in *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995.

[23] Cycorp. (2015) Cycorp - Home of smarter solutions. Zuletzt besucht: 30.10.2015. [Online]. Available: www.cyc.com

[24] C. Schlenoff, R. Ivester, D. Libes, P. Denno, and S. Szykman, "NISTIR 6301: An Analysis of Existing Ontological Systems for Applications in Manufacturing and Healthcare," in *NIST Technical Report*, 1999.

[25] J. Cheng, M. Gruninger, R. D. Sriram, and K. H. Law, "Process specification language for project information exchange," *International Journal of Information Technology in Architecture, Engineering, and Construction*, vol. 1, pp. 307–328, 2003.

[26] K.-Y. Kim, Y. Wang, O. S. Muogboh, and B. O. Nnaji, "Design formalism for collaborative assembly design," *Computer Aided Design*, no. 36, pp. 849–871, 2004.

[27] D. G. Manley, "Assembly Differentiation In CAD Systems," Master's thesis, Industrial Engineering, University of Pittsburgh, 2006.

[28] S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, Eds., *A Port Ontology For Automated Model Composition*. Carnegie Mellon University Pittsburgh, PA 15232, U.S.A. and G.W. Woodruff School of Mechanical Engineering Georgia Institute of Technology Atlanta, GA 30332, U.S.A., 2003.

[29] P. Mijksenaar and P. Westendorp, *Open Here: The Art of Instructional Design*. Joost Elffers Books, New York, 1999.

[30] M. Agrawala, D. Phan, J. Heiser, J. Haymaker, J. Klingner, P. Hanrahan, and B. Tversky, "Designing Effective Step-By-Step Assembly Instructions," Stanford University, Tech. Rep., 2003.

[31] Object Management Group. (2015) Unified Modeling Language (UML) Version 2.5. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.omg.org/spec/UML/

[32] M. Merdan, "Knowledge-based Multi-Agent Architecture Applied in the Assembly Domain," Ph.D. dissertation, Technische Universität Wien, 2009.

[33] P. Inc. (2015) PTC Creo. [Online]. Available: http://de.ptc.com/product/creo

[34] G. Vasilakis and O. Anderson, *Geometric Modelling, Numerical Simulation, and Optimization*. Springer, 2007, ch. Building an Ontology of CAD Model Information, pp. 11–40.

[35] (2015) LDraw.org Centralised LDraw Resources. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.ldraw.org/article/13.html

[36] S. Seedorf, F. F. Informatik, and U. Mannheim, "Applications of Ontologies in Software Engineering," in *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006*, 2006.

[37] A. G. Perez, "A survey on ontology tools," Projekt IST-2000-29243, OntoWeb, 2002, Tech. Rep., 2002.

[38] Standford University. protégé - A free, open-source ontology editor and framework for building intelligent systems. Zuletzt besucht: 30.10.2015. [Online]. Available: http://protege.stanford.edu

[39] Sandia National Laboratories. (2013) Jess®, the Rule Engine for the Java Platform. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.jessrules.com

[40] H. Eriksson. (2010) JessTab: Integrating Protégé and Jess. [Online]. Available: http://www.ida.liu.se/~her/JessTab/

[41] Oracle Corporation. (2015) The Java Language. Zuletzt besucht: 30.10.2015. [Online]. Available: https://www.oracle.com/java/index.html

[42] ——. (2014) Java3D. Zuletzt besucht: 30.10.2015. [Online]. Available: https://java3d.java.net

[43] S. Jorthan, "Basisstrukturen für die Anwendung von wissensbasierten Agentensystemen in der Automatisierungstechnik," Master's thesis, Fakultät für Automatisierungs- und Regelungstechnik, TU Wien, 2006.

[44] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. P. Rice. (1998) Open Knowledge Base Connectivity 2.0.3. Zuletzt besucht: 30.10.2015. [Online]. Available: http://www.ai.sri.com/~okbc/spec/okbc2/okbc2.html

[45] H. H. Wang, N. Noy, A. Rector, M. Musen, T. Redmond, D. Rubin, S. Tu, T. Tudorache, N. Drummond, M. Horridge, and J. Seidenberg, "Frames and OWL Side by Side," in *9th International Protégé Conference, Stanford, California, USA*, 2006.

[46] C. Forgy, *A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* Artificial Intelligence 19, 1982, pp. 17–37.

[47] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," Massachusetts Institute of Technology, Tech. Rep., 1960.

[48] Eclipse Foundation. (2015) Eclipse. Zuletzt besucht: 30.10.2015. [Online]. Available: www.eclipse.org

[49] A. Smith. (2013) Bricksmith virtual Lego modeling for your Macintosh. Zuletzt besucht: 30.10.2015. [Online]. Available: http://bricksmith.sourceforge.net

[50] H. Wood. (2013) Holly-Wood.it A Lego® Fan Site. [Online]. Available: http://www.holly-wood.it

[51] R. Griebl. (2005) LDrawed Mercedes-Benz Actros 4150K in 1:13 scale. Zuletzt besucht: 30.10.2015. [Online]. Available: http://news.lugnet.com/cad/dat/models/?n=2128