Ingenieurfakultät Bau Geo Umwelt
Lehrstuhl für Computergestützte Modellierung und Simulation
Prof. Dr.-Ing. André Borrmann

Umsetzung graphbasierter Methoden zur automatisierten Modellerstellung in parametrischen CAD-Werkzeugen

Martin Slepicka

Bachelorthesis

für den Bachelor of Science Studiengang Umweltingenieurwesen

Autor	Martin Slepicka
Matrikelnummer	[redacted]
Betreuer	Prof. Dr.-Ing. André Borrmann
	Simon Vilgertshofer
Ausgabedatum	01. Juni 2019
Abgabedatum	12. November 2019

Abstract

High-quality models are indispensable in all areas of construction planning and execution. It is particularly important that these models are as error-free as possible, otherwise unforeseeable delays may occur during the later construction phase. In the field of infrastructure planning, for example, multi-scale models are used due to the long-stretched character of the structures (sometimes several kilometres), which are available in various Levels of Detail (LODs). In order to support the engineers and designers in the usually very complex creation of such models and to ensure their consistency, a system was developed with which models of this kind can be graphically represented and easily modified in this form using graph replacement rules, so that repetitive modelling processes can be carried out semi-automatically. Such graphical representations should be correctly interpretable for all common Computer Aided Design (CAD) systems, which has so far only been shown for the CAD software Autodesk Inventor. In this thesis further translation tools are developed, which make this interpretation capability also available for the software solutions Siemens NX and FreeCAD.

Zusammenfassung

Hochwertige Modelle sind in allen Bereichen der Bauplanung und -ausführung unerlässlich. Hierbei ist es besonders wichtig, dass diese Modelle weitestgehend fehlerfrei sind, da es sonst bei der späteren Bauausführung zu unvorhersehbaren Verzögerungen kommen kann. Im Bereich der Infrastrukturplanung z.B. werden aufgrund des langestreckten Charakters der Bauwerke (teils mehrere Kilometer) mehrskalige Modelle verwendet, die in verschiedenen Detailstufen, sogenannten Levels of Detail (LODs), vorliegen. Um die Ingenieure und Designer bei der üblicherweise sehr aufwändigen Erstellung solcher Modelle zu unterstützen und um deren Konsistenz zu gewährleisten wurde ein System entwickelt, mit dem sich Modelle dieser Art graphisch darstellen und in dieser Form über Graphenersetzungsregeln einfach modifizieren lassen, so dass repetitive Modellierungsvorgänge halbautomatisiert durchgeführt werden können. Solche graphische Darstellungen sollen für alle gängigen Computer Aided Design (CAD)-Systeme korrekt interpretierbar sein, was bisher aber nur für die CAD-Software Autodesk Inventor gezeigt wurde. In dieser Arbeit werden daher weitere Übersetzungstools entwickelt, welche diese Interpretationsfähigkeit auch für die Softwarelösungen Siemens NX und FreeCAD verfügbar macht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
BIM	Building Information Modeling
CAD	Computer Aided Design
CSG	Constructive Solid Geometry
DLL	Dynamic Link Library
GCS	Geometric Constraint Solver
GUI	Graphical User Interface
IDE	Integrated Development Environment
LOD	Level of Detail

Kapitel 1

Einleitung

Die Grundlage für die erfolgreiche und ressourcenschonende Durchführung von Bauarbeiten jedweder Art stellt ein detailreiches und gut durchdachtes Modell dar. Jedoch können selbst auf den ersten Blick klein erscheinende Modellfehler große zeitliche Verzögerungen beim späteren Bauprozess hervorrufen und so, zusätzlich zu den wirtschaftlichen Schäden, für politische und soziale Spannungen sorgen. Dabei steigt das Fehlerrisiko je komplexer das Projekt und damit auch das Modell ist und umso mehr Fachplaner aus verschiedenen Bereichen daran mitwirken. Zudem steigen die Anforderungen an entsprechende Modelle an, da moderne Methoden und Workflows im Rahmen des Building Information Modeling (BIM) erfordern, dass im Modell nicht nur die Geometrien weitestgehend fehlerfrei vorliegen, sondern dazu auch Semantik enthalten ist. Es können nur mithilfe von hochwertigen Modellen die Vorzüge des BIM voll ausgeschöpft und damit aus den Modellen abgeleitete Planungs-, Bau- und später auch Instandhaltungsprozesse gewinnbringend unterstützt werden.

Im Straßenbau beispielsweise ist es sinnvoll Modelle zu verwenden, die in verschiedenen Detailstufen vorliegen, da solche Bauwerke aufgrund ihres langgestreckten Charakters (teils mehrere Kilometer) über einen sehr großen Skalierungsbereich dargestellt werden müssen. Die verschiedenen Detailstufen müssen hierbei untereinander Konsistent sein, d.h. sämtliche Informationen der niederen Detailstufen müssen auch in den höheren enthalten sein und eventuelle Änderungen am Modell in allen Detailstufen in gleicher Weise angewendet werden. Um dies zu bewerkstelligen werden hierfür parametrische Modelle verwendet, da diese durch entsprechend definierte Zwangsbedingungen bei Änderungen konsistent bleiben. Ein Problem ist allerdings, dass die händische Erstellung solcher parametrischen Modelle ein durchaus langwieriger, komplexer und somit sehr fehleranfälliger Prozess ist (Borrmann et al., 2014).

Als Lösungsansatz für dieses Problem wird von Vilgertshofer und Borrmann (2015, 2016, 2017, 2018) die Möglichkeit zur Nutzung von Graphtransformation vorgestellt, welche eine solche Modellierung deutlich vereinfachen soll, indem repetitive und automatisierbare Arbeitsabläufe formalisiert werden. Dazu wurde zunächst im Rahmen einer Masterarbeit ein Graphensystem,

bzw. das zugrunde liegende Metamodell und Regelsystem, entwickelt (Vilgertshofer, 2014) und seither im Zuge einer Promotion weiterentwickelt. Mit einem solchen Graphen kann dann z.B. ein Tunnelbauwerk repräsentiert werden und später mit entsprechenden Tools halbautomatisch in eine parametrische Darstellung umgewandelt werden. Dabei kann mithilfe von Graphersetzungsregeln, die im Graphensystem formalisiert wurden, das Modell einfach verändert werden, um so z.B. von einer Detailstufe in die andere zu wechseln.

Die Umwandlung soll hierbei für alle gängigen parametrischen CAD-Systeme, wie z.B. Autodesk Inventor, Siemens NX oder FreeCAD, möglich sein. Bisher wurde aber nur gezeigt, dass sich ein entsprechender Graph mit Autodesk Inventor interpretieren lässt. In dieser Arbeit sollen daher weitere Übersetzungstools implementiert werden, welche die CAD-Systemen Siemens NX und FreeCAD um diese Interpretationsfähigkeit erweitert. Hierzu werden zunächst in Kapitel 2 die notwendigen theoretischen Grundlagen erläutert, in Kapitel 3 die verwendete Software beschrieben und in Kapitel 4 die Vorgehensweise bei der Implementierung sowie die Funktionsweise der beiden Übersetzungstools dargestellt. Schließlich werden in Kapitel 5 anhand zweier Beispiele die Interpretationsfähigkeiten der beiden fertiggestellten Tools sowie eventuelle Mängel vorgestellt.

Kapitel 2

Theoretische Grundlagen

Zur besseren Nachvollziehbarkeit der in dieser Thesis bearbeiteten Problemstellung und deren Umsetzung ist es zu aller Erst notwendig einige der in Folge verwendeten Begriffe näher zu erläutern. Im Folgenden wird daher ein grundlegender Einblick in die parametrische Modellierung in CAD-Systemen sowie in die Graphentheorie geschaffen. Dabei wird im jeweiligen Unterkapitel insbesondere auf die für diese Thesis wichtigen Konzepte eingegangen.

2.1 Parametrische Modellierung

Aufgrund der vielfältigen Einsatzmöglichkeiten für Modelle gibt es eine ebenso große Vielfalt an Modellierungswerkzeugen sowie einige verschiedene Modellierungsmethoden. Als sehr einfache und schnelle Methode ist hier die **direkte Modellierung** zu nennen. Hierbei wird das Modell in seinen exakten Maßen aus geometrischen Grundelementen gezeichnet, z.B. Punkte, Linien oder Kreisbögen, die zusammengesetzt Flächen und Körper ergeben. Diesen geometrischen Elementen wird hierbei allerdings keine weitere Information mitgegeben, daher müssen bei jeder Änderungen am Modell alle betroffene Element einzeln angepasst werden.

Aufbauend zur direkten Modellierung wird bei der **parametrischen Modellierung** das Modell um sogenannte Parameter erweitert, welche hier als veränderbare Größen, die Eigenschaften und Abhängigkeiten in und zwischen Modellen beschreiben, zu verstehen sind. Dabei können Parameter untereinander Beziehungen aufweisen, d.h. sie können arithmetische, logische oder geometrische Abhängigkeiten aufweisen (Vajna et al., 2018, Kapitel 5.5). Es gibt verschiedene Arten von Parametern, solche die z.B. physikalische Eigenschaften festlegen oder Daten für Fertigungsprozesse liefern. In dieser Thesis sind aber nur **Geometrieparameter**, welche die Gestaltungslogik des Modells festlegen, wichtig.

2.1.1 Zwangsbedingungen

Mithilfe von Geometrieparametern können **Zwangsbedingungen** (engl. *constraints*) definiert werden, mit denen die Position oder die Abhängigkeiten geometrischer Elemente im Modell festgelegt werden können. Jede angewendete Zwangsbedingung verringert dabei die Anzahl der Freiheitsgrade eines geometrischen Objekts um die Anzahl seiner Valenzen (Parameterbeziehungen). Sind alle Freiheitsgrade durch Zwangsbedingungen beseitigt, gilt das Objekt als vollständig parametrisiert (engl. *fully constrained*) und ist in seiner Form fixiert. Ein Satz an Zwangsbedingungen kann jedoch mehr als nur eine Geometrie beschreiben. Wird also nicht sorgfältig modelliert, kann es sein, dass eine unerwünschte Form fixiert wird und die Bedingungen neu gesetzt werden müssen – mehr dazu wird in Abschnitt 2.1.4 vorgestellt.

Ein parametrisiertes Objekt, welches noch Freiheitsgrade besitzt, gilt als **unterbestimmt** und ist damit ins seiner Form nicht fixiert. Bei Parameteränderungen bleibt hierbei die gewünschte Form in der Regel nicht erhalten. Werden an einem Objekt mehr Zwangsbedingungen angewandt als nötig, gilt es als **überbestimmt** und ist damit nicht mehr darstellbar, wenn Bedingungen sich gegenseitig widersprechen. Um ein in sich konsistentes parametrisches Modell zu erstellen, ist es daher wichtig genau alle Freiheitsgrade einzuschränken.

Zwangsbedingungen lassen sich in zwei Klassen eingeteilen, in geometrische und dimensionale Zwangsbedingungen. **Geometrische Bedingungen** definieren dabei die Anordnung der Elemente eines Objekts im Raum. Ein Beispiel ist hierbei die Zwangsbedingung *Horizontal*, hiermit wird im 2D-Raum die betroffene Linie parallel zu der x-Achse des Koordinatensystems angeordnet. **Dimensionale Bedingungen** legen dagegen Abstände und Maße in einem Modell fest und beschreiben damit die Größe eines Objekts. In Tabelle 2.1 werden die in dieser Arbeit verwendeten Zwangsbedingungen aufgelistet und kurz erläutert – hier werden die englischen Begriffe gelistet, nachdem in dieser Arbeit englischsprachige Programme genutzt werden.

Tabelle 2.1: Wichtige geometrische und dimensionale Zwangsbedingungen.

Constraint	**Bedeutung**
coincident	Setzt die Koordinaten zweier Punkte gleich.
concentric	Setzt die Mittelpunkte zweier Kreise koinzident.
parallel	Setzt zwei Linien zueinander parallel.
perpendicular	Stellt eine Linie zu einer Anderen senkrecht.
collinear	Legt fest, dass zwei Linien auf der selben Geraden liegen.
tangent	Legt fest, dass sich zwei Kurven berühren aber nicht schneiden.
horizontal	Richtet eine Linie horizontal aus.
vertical	Richtet eine Linie vertikal aus.
fixed	Fixiert ein Element im Bezug zum Koordinatensystem.
linear dimension	Legt den linearen Abstand zwischen zwei Punkten fest.
radial dimension	Legt den Radius von Kreisen oder Bögen fest.

Tabelle 2.2: Mögliche Ports für einfache Linienelemente

Geometrie	**mögliche Ports**
Punkt	–
Linie	Startpunkt, Mittelpunkt, Endpunkt, Punkt auf Linie
Kreis	Mittelpunkt, Punkt auf Kreisbogen
Kreisbogen	Startpunkt, Mittelpunkt, Endpunkt, Punkt auf Kreisbogen

Für zwei dieser Bedingungen, *coincident* und *linear dimension* (zwischen zwei Geometrieelementen), bedarf es neben der Angabe der einzuschränkenden Geometrien noch der zusätzlichen Information an welchen Punkten „angedockt“ werden soll. Im folgenden werden diese Andockpunkte **Ports** genannt. Z.B. die *coincident*-Bedingung legt aufeinander fallende Punkte fest und bei einer Anwendung dieser Bedingung auf zwei Linien muss zusätzlich festgelegt werden, welche Punkte auf den beiden Linien zusammenfallen sollen. Tabelle 2.2 zeigt eine Liste der für die entsprechenden Skizzen-Elemente verfügbaren Ports.

2.1.2 Datenstruktur eines parametrischen Modells

Mit den in dieser Arbeit verwendeten CAD-Systeme erzeugt der Modellierer beim Design eines Objekts je nach Komplexität ein oder mehrere Baugruppen- und Bauteildateien. Ein Bauteil (engl. *Part*) stellt dabei ein Objekt dar, das nicht zerstörungsfrei in seine Einzelkomponenten zerlegt werden kann, während eine Baugruppe (engl. *Assembly*) ein zusammengesetztes Produkt darstellt, bestehend aus ein oder mehreren Bauteilen oder auch Unterbaugruppen.

Es ergibt sich damit eine hierarchische Datenstruktur, an dessen oberster Ebene sich die **Baugrauppe** befindet (vgl. Abb. 2.1). In Baugruppen werden Bauteile oder untergeordnete Baugruppen positioniert und über Baugruppenbeziehungen (engl. *assembly constraints*) miteinander verknüpft.

Auf der zweiten Hierarchieebene befindet sich das **Bauteil**, in dem ein oder mehrere geometrische Körper zu einem Objekt zusammengefügt werden. Die Körper werden dabei entweder

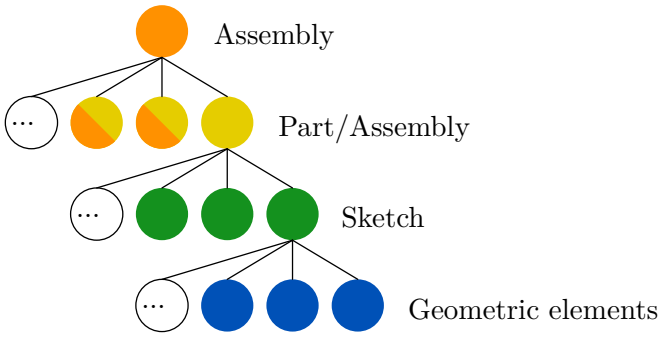


Abbildung 2.1: Hierarchische Datenstruktur eines parametrischen Modells.

über boolsche Operationen miteinander kombiniert oder über Bezugsebenen (engl. *Planes*) aneinander positioniert und sind entweder geometrische Primitive (Quader, Kugel, Kegel, etc.) oder Extrusions- bzw. Rotationskörper, denen je eine zweidimensionale Skizze (engl. *Sketch*) zugrunde liegt.

Demnach befindet sich auf der dritten Hierarchieebene die **Skizze**. In diesen 2D-Zeichnungen wird die Kontur des zu erstellenden Körpers festgelegt. Dazu werden geometrische Elemente mithilfe von Zwangsbedingungen (siehe Abschnitt 2.1.1) so zueinender angeordnet, dass sich die gewünschte Form ergibt. Skizzen werden über oben genannte Bezugsebenen im Bauteil ausgerichtet und können z.B. entlang der Bezugsnormalen extrudiert (engl. *Extrusion*) oder entlang einer Kurve (auch *Alignment*) gezogen werden (engl. *Sweeping*).

Auf der vierten und untersten Hierarchieebene stehen damit die **geometrischen Elemente**. Zu diesen Elementen gehören Punkte, Linien, Kreise und Kreisbögen sowie Projektionen davon (Abbilder dieser Elemente aus anderen Skizzen). In den meisten CAD-System gibt es noch weitere solcher Grundelemente, in dieser Arbeit sind jedoch nur die eben genannten von Bedeutung.

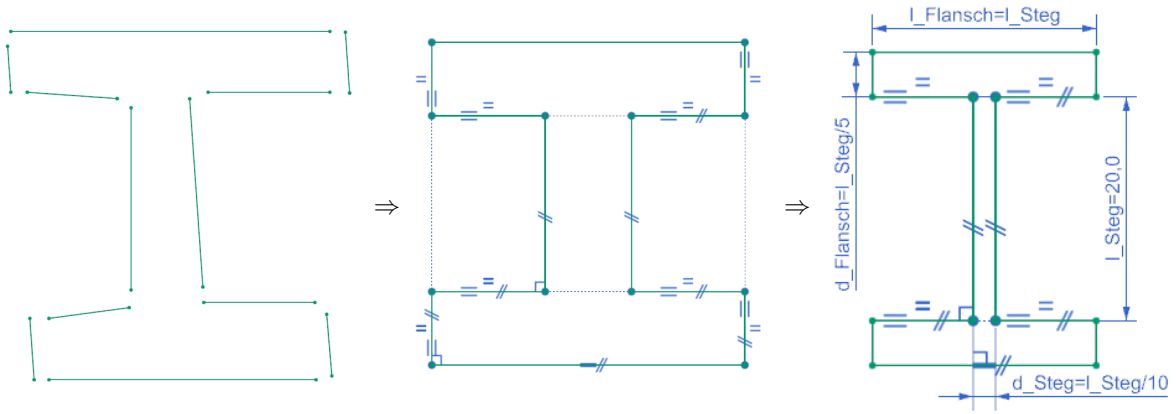
2.1.3 Methodik bei der parametrischen Modellierung

Die beiden in dieser Arbeit entwickelten Übersetzungstools sollen in der Lage sein ein parametrisches Modell zu erstellen. Daher ist es nötig einen allgemein gültigen Arbeitsablauf der Modellierung darzustellen. Hierzu muss grundsätzlich als erstes die oberste Hierarchiestufe erstellt werden, d.h. die Baugruppen-Datei wird erzeugt – bei einfachen Modellen, die nur aus einem Bauteil bestehen kann die Baugruppen-Ebene wegfallen und die Bauteil-Datei als oberste Instanz angenommen werden.

Nach und nach werden dann alle Bauteile erstellt und hinzugefügt. Hierzu wird für jedes Bauteil eine eigene Bauteil-Datei erzeugt und darin die entsprechende Geometrie definiert. In der Regel wird der 3D-Körper des Bauteils über 2D-Skizzen beschrieben, also werden hier der Reihe nach alle nötigen Skizzen für das Bauteil erstellt.

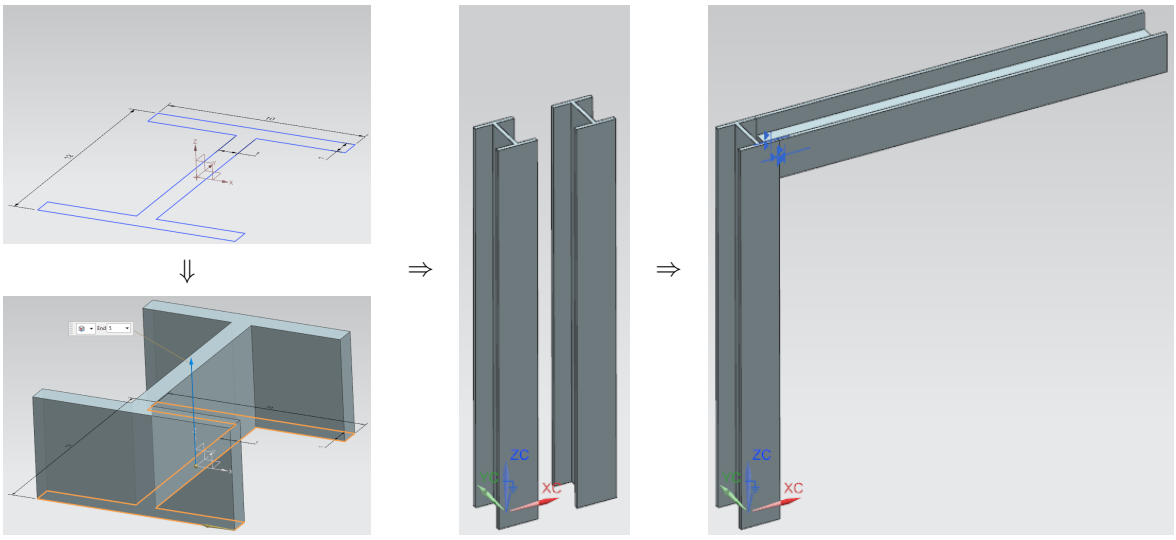
Innerhalb einer Skizze werden dann alle geometrischen Elemente angelegt. Diese müssen nicht zwangsläufig in der richtigen Position erstellt werden sollten aber in sinnvoller Ausrichtung angelegt werden, um den nächsten Schritt zu erleichtern. Als nächstes werden die Zwangsbedingungen definiert. Hierbei gibt es keine zwingend vorgeschriebene Reihenfolge, sind die Geometrien aber nicht sinnvoll angeordnet, kann es passieren, dass nach dem Anlegen einer Bedingung nicht die gewünschte Form erzielt wird (siehe auch Abschnitt 2.1.4).

In Abb. 2.2 wird am Beispiel einer Skizze für einen IPE-Träger eine sinnvolle Vorgehensweise dargestellt. Zuerst werden die Linien gezeichnet, dabei wird zur Veranschaulichung nicht darauf geachtet, dass die Linienenden entsprechend zusammenfallen. Das Polygon wird dann mithilfe



(a) geometrische Elemente (b) geometrische Einschränkungen (c) dimensionale Einschränkungen

Abbildung 2.2: Beispielhafte Vorgehensweise beim erstellen eines Sketches für einen IPE-Träger.



(a) Extrusion aus Sketch (b) Baugruppe (c) Baugruppeneinschränkung

Abbildung 2.3: 3D-Gestaltung zweier IPE-Träger mit anschließender Anordnung im Raum.

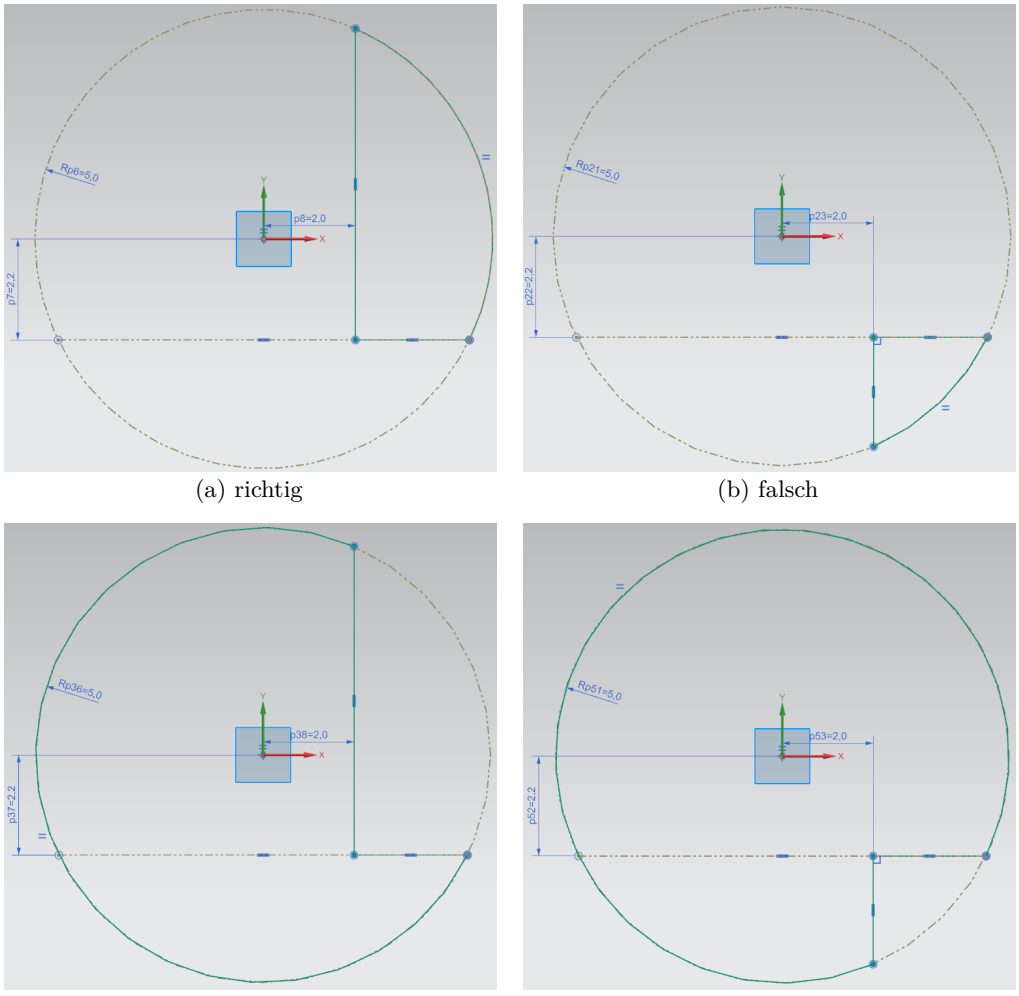
von *coincident*-Bedingungen geschlossen (im Bild durch große blaue Punkte an den Linienenden symbolisiert) und über weitere geometrische Zwangsbedingungen in seiner Form definiert. Letztlich wird nur noch die Größe durch anbringen von dimensionalen Zwangsbedingungen festgelegt. In Abb. 2.2c ist damit eine vollständig parametrisierte Skizze dargestellt. Die hier aufgeführte Reihenfolge beim Anbringen der Zwangsbedingungen muss allerdings nicht verpflichtend genauso eingehalten werden.

Sobald eine Skizze fertiggestellt ist kann aus ihr z.B. per Extrusion ein 3D-Körper erstellt werden. Hierzu wird nur das entsprechende Extrusionstool (bzw. Rotations-, Sweep-Tool, etc.) auf die Skizze angewendet und mit den entsprechenden Parametern (z.B. die Extrusionsdistanz) die Größe des Körpers definiert (vgl. Abb. 2.3a). Wenn danach ein zweites Bauteil erstellt wird, so wird hierzu genauso eine neue Bauteil-Datei erstellt, diese der Baugruppe hinzugefügt

und die Geometrie erstellt. Abbildung 2.3b zeigt dabei eine Baugruppe mit zwei versetzt zueinander eingefügten Bauteilen, die daraufhin mittels Baugruppenbeziehungen zueinander ausgerichtet werden können (vgl. Abb. 2.3c). Baugruppenbeziehungen werden in dieser Arbeit allerdings nicht weiter behandelt.

2.1.4 Geometric Constraint Solver

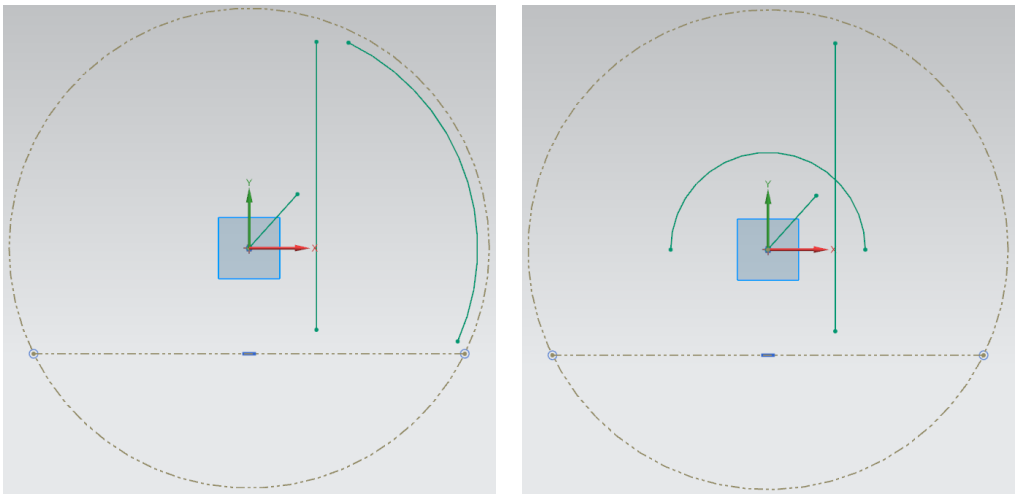
Bisher wurde nur erläutert was Zwangsbedingungen sind und wann sie angewendet werden. Im Folgenden wird zudem erläutert wie das CAD-System Zwangsbedingungen umsetzt. Da in der Regel Verschiebungen der Geometrien notwendig sind, wenn eine neue Zwangsbedingung definiert wird, muss das CAD-System hierfür einen bestimmten Lösungsmechanismus besitzen. Genau dieser Mechanismus wird als Geometric Constraint Solver (GCS) bezeichnet. Dieser Solver erzeugt nach jeder neuen Definition oder Veränderung einer Bedingung aus den vorhan-



(a) richtig (b) falsch

(c) falsch (d) falsch

Abbildung 2.4: Realisierungsmöglichkeiten desselben Graphen ohne Beachtung der initialen Positionierung der Elemente – nur eine Auswahl aller Möglichkeiten.



(a) sinnvoll (b) nicht sinnvoll

Abbildung 2.5: Initiale Positionierung der einzelnen Elemente.

denen Geometrien und Zwangsbedingungen eine neue Anordnung, die alle Einschränkungen erfüllt.

Wie schon in Abschnitt 2.1.1 beschrieben kann ein vollständiger Satz von Zwangsbedingungen verschiedene Anordnungen der Geometrien beschreiben, ebenso kann aber auch eine Anordnung durch verschiedene Zwangsbedingungssätze dargestellt werden. Das Problem daran ist, dass beim sequenziellen Anbringen der Bedingungen Verschiebungen der Geometrien auftreten können, die nicht zum richtigen Ergebnis oder im schlimmsten Fall sogar zu unlösbaren Zwischenergebnissen führen. Ein vollständiger Zwangsbedingungssatz ist hier als solcher zu verstehen, der zu einem vollständig parametrisierten Modell führt.

Abbildung 2.4 veranschaulicht das Problem der Mehrdeutigkeit von Zwangsbeziehungssätzen. Bei allen vier Möglichkeiten sind die entsprechenden Geometrien auf die selbe Art und Weise verknüpft, der einzige Unterschied besteht in der anfänglichen Platzierung der einzelnen Elemente (vgl. Abb. 2.5). Während die Positionierung in Abb. 2.5a die richtige Lösung liefern sollte, erzielt die Positionierung in Abb. 2.5b je nach Implementierung des GCS sehr wahrscheinlich eine der falschen Lösungen. Problematisch ist allerdings auch, dass nicht mit 100%-iger Sicherheit gesagt werden kann, dass alle unterschiedlichen Solver das gleiche Ergebnis liefern.

Robuste GCS können in der Regel in Abhängigkeit zur Ausgangssituation die Lösung finden, für welche die kleinste Gesamtverschiebung benötigt wird. Eine sinnvolle initiale Positionierung der Geometrieelemente ist daher essentiell für die Durchführung einer automatisierten parametrischen Modellierung.

2.1.5 Koordinatensysteme

In Abschnitt 2.1.2 wurden die verschiedenen Hierarchieebenen vorgestellt. Dabei ist zusätzlich zu erläutern, dass die Strukturen Baugruppe, Bauteil und Skizze jeweils ihren eigenen geometrischen Raum aufspannen mit jeweils einem eigenen Koordinatensystem. Da diese Strukturen ineinander verschachtelt sind, stehen diese Koordinatensysteme zueinander in Relation.

Eine Baugruppe ist nach Definition eine Sammlung und Anordnung von Bauteilen, dabei kann dasselbe Bauteil auch mehrmals aber in unterschiedlicher Positionierung eingebettet werden. Die Position der Bauteile wird hierzu im Koordinatensystem der Baugruppe festgelegt, die Positionen der jeweiligen Bauteilgeometrien werden dagegen im Koordinatensystem der Bauteile festgelet. In einer solchen Situation spricht man von **globalen** und **lokalen** Koordinatensystemen, dabei ist das Koordinatensystem der Baugruppe relativ gesehen zum Koordinatensystem der Bauteile global und umgekehrt das der Bauteile relativ zur Baugruppe lokal. Genauso verhält es sich mit den Koordinatensystemen von Skizzen in Bezug auf das entsprechende Bauteil.

In den meisten CAD-Systemen können neben den Standard Koordinatensystemen (Ursprung von Baugruppe, Bauteil oder Skizze) noch weitere lokale Koordinatensysteme definiert werden. Dies macht genau dann Sinn, wenn gewisse Substrukturen in z.B. einem Bauteil relativ zu anderen Substrukturen platziert werden sollen. In dieser Arbeit werden jedoch nur die Standard-Koordinatensysteme verwendet, es werden also alle Geometrien anhand des Ursprungs der jeweiligen Struktur ausgerichtet, in der sie gezeichnet werden.

2.2 Graphen

Eine der Hauptaufgaben dieser Arbeit ist es ein bestehendes Graphensystem zu interpretieren und in ein parametrisches Modell zu übersetzen. Dazu wird an dieser Stelle zunächst grundlegend auf den theoretischen Hintergrund von Graphen eingegangen. Darauffolgend wird das verwendete Graphensystem anhand des implementierten Metamodells beschrieben. Die zur Erstellung des Systems verwendete Software *GrGen.NET* und deren wichtigsten Funktionen wird in Abschnitt 3.1 vorgestellt.

2.2.1 Graphentheorie

Ein Graph stellt ein abstraktes mathematisches Modell dar das für die Analyse oder Beschreibung von netzartigen Strukturen, wie zum Beispiel einen Stammbaum oder ein Verkehrsnetz, eingesetzt wird. Formalisiert wird ein Graph durch die Graphentheorie, einem Teilgebiet der Mathematik. Auch wenn die Form verschiedener Graphen sich erheblich unterscheiden kann

ist der grundlegende Aufbau hierbei immer derselbe. Diese Beschreibung und die folgenden Definitionen basieren auf (Tittmann, 2019).

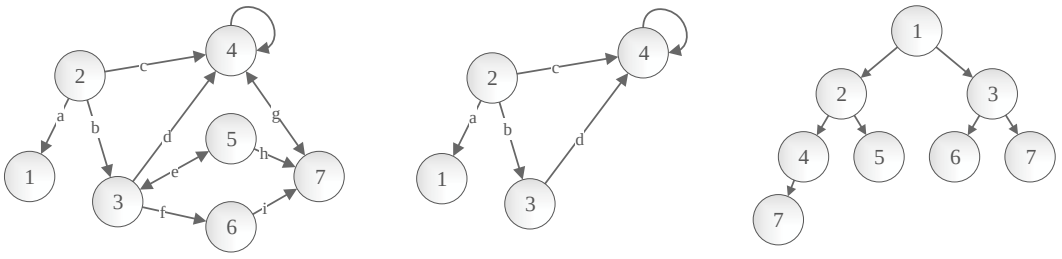
Graphen lassen sich stets als geordnetes Paar von zwei verschiedenartigen Mengen, **Knoten** (engl. *Vertices*) und **Kanten** (engl. *Edges*), darstellen – mathematisch formalisiert: $G = (V, E)$. Hierbei beschreiben die Knoten V in der Regel reale Objekte, wie z.B. geometrische Elemente, und die Kanten E deren Beziehung zueinander, wie z.B. „senkrecht aufeinander“.

Kanten symbolisieren immer eine Verbindung zwischen genau zwei Knoten und gelten als **ungerichtet**, wenn der Kante keine eindeutige Richtung zugewiesen ist. Eine Kante e mit der Eigenschaft $e = \{u, v\} = \{v, u\}$ gilt demnach als ungerichtet, hierbei stellen u und v jeweils einen Endknoten der Kante e dar und werden damit auch als **inzident** zu e bezeichnet. Knoten u, v die über eine Kante e miteinander verbunden sind, werden auch als **adjazent** zueinander bezeichnet.

Ist die Richtung der Kante e dagegen festgelegt, wird sie als **gerichtete** Kante bezeichnet und als geordnetes Paar, $e = (u, v)$ mit Anfangsknoten u und Endknoten v, formalisiert. Gibt es mehrere Kanten e_i mit denselben Anfangs- und Endknoten u, v, so werden diese als **Mehrfachkanten** bezeichnet. Hat eine Kante denselben Anfangs- wie Endknoten, so spricht man dabei von einer Schlinge oder auch **Schleife**.

Ein Graph, der mindestens eine gerichtete Kante besitzt, wird als gerichteter Graph bezeichnet, ansonsten als ungerichteter Graph. Das in dieser Arbeit verwendete Graphensystem verwendet gerichtete Kanten, Mehrfachkanten und Schleifen, daher wird ein solcher Graph auch **gerichteter Multigraph** genannt. Ist ein Graph A vollständig Bestandteil eines anderen Graphen B, so ist A ein **Subgraph** zu B und umgekehrt B der **Supergraph** zu A.

Dargestellt werden Graphen üblicherweise indem die Knoten als Formen, wie z.B. Kreise oder Rechtecke, und die Kanten als Linien oder Pfeile zwischen den Knoten gezeichnet werden. Beispiele hierfür sind in Abb. 2.6 gegeben, dabei wurden nur gerichtete Graphen dargestellt. Besonders hervorzuheben ist hier der Graph in Abb. 2.6c, eine solche Struktur wird auch als Baum bezeichnet, hier sogar noch spezieller als Binärbaum, da nur jeweils zwei Kanten in die nächst untere Ebene führen. Ein weiteres Beispiel für einen Baum ist Abb. 2.1, hierzu kann



(a) gerichteter Graph (b) Subgraph von 2.6a (c) Binärbaum

Abbildung 2.6: Übliche Darstellungsweise von Graphen anhand typischer Beispiele

gesagt werden, dass mit Bäumen hierarchische Zusammenhänge sehr gut dargestellt werden können. Der Knoten in der obersten Ebene, der also keine Kante in eine nächst höhere Ebene aufweist, wird hier als **Wurzelknoten** bezeichnet.

Knoten und Kanten sind wie bisher beschrieben sehr abstrakte Konstrukte, die Objekte und deren Beziehungen beschreiben. Komplexe Netzstrukturen können aber nicht unbedingt mit diesem Level an Abstraktion dargestellt werden. Es ist aber möglich die Knoten und Kanten ganz analog zur objektorientierten Programmierung in verschiedene Typen (**Klassen**) einzuteilen und sie mit Eigenschaften (**Attribute**) zu versehen. Hierzu ist allerdings ein beschreibendes Modell, ein sogenanntes **Metamodell**, notwendig. In einem solchen Metamodell werden die verschiedenen Knoten- und Kantentypen und ihre Attribute definiert. Ein Beispiel hierfür ist in Abschnitt 2.2.3 aufgeführt.

2.2.2 Graphersetzung

Das in dieser Arbeit verwendete Graphensystem soll zur Darstellung von Modellen in mehreren Detailstufen genutzt werden und muss daher in der Lage sein, teilweise große Veränderungen am Graphen (Transformationen) durchzuführen. Diese Änderungen können hierbei in sogenannten Graphersetzungsregeln festgehalten und damit automatisiert angewendet werden.

Eine Graphersetzung beschreibt dabei den genauen Vorgang, der von einem bestimmten Ausgangsgraphen zu einem bestimmten Zielgraphen führt. Die bei der Transformation angewendete Graphersetzungsregel definiert indes welche Teile des Ausgangsgraphen auf welche Art und Weise verändert, ersetzt oder gelöscht werden sollen. Der zu verändernde Teilgraph wird als **Mustergraph** (engl. *pattern graph*) bezeichnet und durch Anwendung der Regel in die Zielform, den **Ersetzungsgraph** (engl. *rewrite graph*), transformiert. Als grundlegendes Prinzip wird in dem hier verwendeten Graphensystem der **Single Pushout Approach** angewendet (siehe auch Blomer, Geiß & Jakumeit, 2013; Grzegorz, 1999; Heckel, 2006).

2.2.3 Verwendetes Metamodell

Das in dieser Arbeit verwendete Metamodell zur Definition der Knoten- und Kantentypen wurde im Rahmen einer Masterarbeit entwickelt und wird seither im Zuge einer Promotion weiterentwickelt. Es ist so ausgelegt, dass sich die in Abschnitt 2.1.3 vorgestellte Datenstruktur perfekt graphisch abbilden lässt (vgl. auch Abb. 2.1). Knoten stellen in diesem Modell reale geometrische Elemente und Sammelstrukturen (Assembly, Part, Sketch) dar, wohingegen Kanten unter anderem die Zwangsbedingungen symbolisieren (vgl. Abb. 2.7).

Die Knoten lassen sich hierbei in zwei abstrakte Unterklassen einteilen, zum einen in die prozeduralen- und zum anderen in die Skizzenelemente. Dabei stellen die **prozeduralen Elemente** die genannten Sammelstrukturen, 3D-Körper und boolsche Operationen dar.

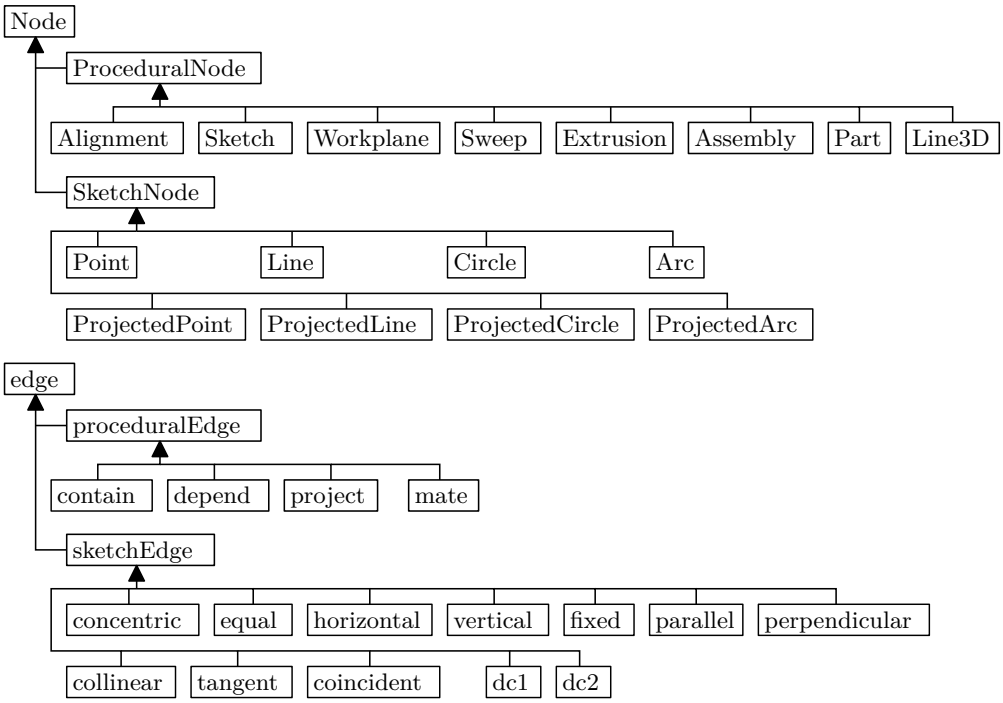


Abbildung 2.7: Metamodell des verwendeten Graphensystems ohne Attribute. Nachgebildet nach (Vilgertshofer & Borrmann, 2015)

Wohingegen die **Skizzenelemente** ausschließlich 2D-Geometrien und deren Projektionen darstellen. Da Objekte auch bestimmte Eigenschaften besitzen müssen, wie z.B. die Position oder Orientierung im Raum, sind jedem Knoten zudem Attribute zugewiesen, um diese Informationen abzuspeichern.

Beispielsweise werden einer Linie folgende optionale Informationen mitgegeben: **Temporäre Koordinaten** des Start- und Endpunktes, Name und ein boolscher Wert, der anzeigt ob die Linie eine Hilfslinie darstellen soll. Generell wird die endgültige Geometrie durch die Zwangsbedingungen bestimmt, daher sind die endgültigen Positionen der einzelnen Elemente nicht zwingend im Vorfeld verfügbar. Aber wie schon im Abschnitt 2.1.4 beschrieben ist es notwendig die einzelnen Elemente vorab sinnvoll zu platzieren bevor die Zwangsbedingungen angebracht werden um sicherzustellen, dass der GCS die gewünschte Lösung liefert. Für dieses vorläufige Platzieren werden dann die temporären Koordinaten eingesetzt.

Weitere besonders hervorzuhebenden Attribute sind vom Typ *Port* und *Plane*. Wie schon in Abschnitt 2.1.1 beschrieben brauchen manche Zwangsbedingungen die Information wo sie andocken müssen. Diese Information wird durch die Attribute `source_port` und `target_port` im Graphensystem gespeichert. Für die beiden Attribute wurde ein sogenannter *Enumerator* mit dem Namen `Port` definiert, der alle in Tabelle 2.2 genannten möglichen Ports beziffert. In gleicher Weise wird auch ein *Enumerator* für Ebenentypen eingeführt, welcher verschiedene Ausrichtungen einer Ebene beziffert und für das Attribut `type` der Klasse `WorkPlane` eingesetzt

wird. In diesem Graphensystem werden nur drei Ausrichtungen definiert: *yz*, *xz* un *xy*. Die Buchstaben zeigen dabei an welche Koordinatenachsen die Ebene aufspannen, dabei zeigt der Normalenvektor der entsprechenden Ebene in Richtung des Kreuzproduktes der Achsen (z.B. für *xy* in Richtung des Vektors $\mathbf{n} = \mathbf{x} \times \mathbf{y}$)

Genau wie bei den Knoten lassen sich auch die Kanten in zwei abstrakte Unterklassen aufteilen. Darunter sind zum einen die prozeduralen Kanten und zum anderen die Skizzen-Kanten. Die **prozeduralen Kanten** sind hierbei als Abhängigkeitsbeziehungen zu verstehen wohingegen die **Skizzen-Kanten** die Zwangsbedingungen unter den geometrischen Elementen darstellen. Auch diese Beziehungen können wie die geometrischen Objekte zusätzliche Eigenschaften besitzen, wie z.B. das Längenmaß bei den dimensionalen Einschränkungen. Daher werden im Metamodell auch den Kanten Attribute zugewiesen.

Die **Abhängigkeitsbeziehungen** geben in erster Linie bei der Erstellung der CAD-Geometrie eine zeitliche Reihenfolge vor, in der die einzelnen Elemente erstellt werden müssen. Nur so kann die in Abschnitt 2.1.3 vorgestellte Datenstruktur eingehalten werden. Die verschiedenen Unterklassen der prozeduralen Kanten geben der entsprechenden Abhängigkeitsbeziehung noch weitere Bedeutung. Der Kantentyp `contain` zeigt an, dass das Ziel im Ursprungsknoten enthalten ist (nur anwendbar auf Skizzen und dessen Elemente), dagegen weist der `project` Kantentyp darauf hin, dass der Ursprungsknoten projiziert werden soll. Andere derartige Kantentypen können z.B. Baugruppeneinschränkungen sein.

Die **Zwangsbedingungen** geben keine unbedingte zeitliche Reihenfolge an, es ist hier nur wichtig, dass beide zur Kante inzidente Knoten bereits erstellt wurden, bevor die Einschränkung angewendet wird. Der entsprechende Kantentyp definiert hier die Art der Zwangsbedingung.

Kapitel 3

Verwendete Software

Wie zuvor beschrieben, sollen in dieser Arbeit ähnlich zu dem schon existierenden Übersetzungstool für Autocad Inventor zwei weitere gleichwertige Tools erstellt werden. Zum einen für das kommerzielle CAD System Siemens NX und zum anderen für das frei erhältliche CAD System FreeCAD. Beide Tools sollen auf gleiche Art und Weise einen bestehenden Graphen interpretieren und in eine parametrische Darstellung übersetzen. Es werden hierzu zunächst die Graphersetzungs-Software und dann die jeweiligen CAD Systeme vorgestellt.

Zur Programmierung dieser beiden Tools werden die Programmiersprachen C# und Python verwendet. Für die Sprache C# wird dabei das Integrated Development Environment (IDE) *Visual Studio* der Firma Microsoft verwendet und für Python das IDE *pyCharm*, entwickelt von JetBrains. Die hierfür verwendeten Bibliotheken werden im folgenden für das jeweilige CAD System beschrieben.

3.1 GrGen.NET

Die in dieser Arbeit ausgewerteten Graphen werden mit der **Graphersetzungssoftware GrGen.NET** erstellt und transformiert. GrGen.NET liegt derzeit in der Version 4.5[1] vor und wird seit 2003, anfangs als Teil einer Diplomarbeit (Hack, 2003), an der Universität Karlsruhe entwickelt. GrGen.NET ist ein Software-Entwicklungstool das für die Handhabung graphartiger Datenstrukturen optimierte Programmiersprachen bereitstellt und damit die Erstellung und Veränderung solcher Datenstrukturen deutlich vereinfacht (Blomer et al., 2013).

Die bereitgestellten Programmiersprachen sind zum einen die **Graph Model Language**, mit der sich ein Metamodell für einen Graphen beschreiben lässt. Mit dieser Sprache werden

[1] Release date: 9. April 2017

Klassen und Unterklassen von Knoten und Kanten sowie deren Attribute definiert und in sogenannten **graph model description Dateien** (`*.gm`) abgespeichert.

Graphersetzungsregeln werden mit der **Rule and Computations Language** definiert und in **rule set Dateien** (`*.grg`) gespeichert. Die Anwendung dieser Regeln wird mit der **Rule Application Control Language** gesteuert, was in dieser Arbeit aber nicht weiter von Bedeutung ist.

Zuletzt ist noch die **Shell Language** zu nennen, welche eine Sammlung von Kommandozeilen-Befehlen ist, mit denen sich das Kommandozeilenprogramm `GrShell.exe` steuern lässt und sich vorgefertigte **Skript Dateien** (`*.grs`) schreiben lassen. Aber da GrGen.NET in dieser Arbeit hauptsächlich über dessen API angesprochen wird, ist auch die Shell Language an dieser Stelle nicht von großer Bedeutung.

Ausgehend von den *graph model description-* und *rule set*-Dateien werden über das Compilerprogramm `grgen.exe` die beiden .NET assemblies `*Model.dll` und `*Actions.dll` erstellt, die unter anderem für jede in den model Dateien definierten Knoten- oder Kantentypen und für jede in den rule Dateien definierte Graphersetzungsregel je ein **Interface** bereitstellen. Über diese Interfaces können im Programmcode dann direkt die Attribute der Knoten und Kanten angesprochen werden.

Bei der Entwicklung einer eigenen Software kann, sofern die beiden oben genannten generierten sowie die beiden mitgelieferten DLL-Dateien, `LibGr.dll` und `lgspBackend.dll`, in das Projekt eingebunden werden, auf den vollen Funktionsumfang von GrGen.NET sowie damit erstellte Graphen zugegriffen werden.

3.2 Siemens NX

NX ist eine Produktenwicklungslösung der Firma *Siemens Digital Industries Software*, die ein sehr umfangreiches und leistungsstarkes Toolset zur Verfügung stellt. Dabei umfasst das Toolset Softwarelösungen für die konzeptionelle Konstruktion, disziplinübergreifende Simulation und zur Teilefertigung. Mit dieser Software kann in sehr hoher Qualität vollständig dreidimensional gezeichnet und parametrisch modelliert werden (»NX. Answers for Industry.« 2013). NX läuft auf MacOS, Linux und Windows und ist seit NX4 als 64-Bit Version verfügbar.

Ursprünglich, seit Anfang der 1970er, wurde die Software von der Firma *United Computing* (später *UGS Corporation*) unter dem Namen **Unigraphics** entwickelt und später kombiniert mit der Software **I-DEAS** als „Next Generation“ Version (NX) weitergeführt. 2007 kaufte Siemens die Firma UGS Corporation und änderte den Firmennamen in *Siemens PLM Software* (»Siemens Closes Acquisition of UGS«, 2007) und später in *Siemens Digital Industries Software*.

Für diese Arbeit wird NX in der Version 12[2] verwendet, jedoch ist der benötigte Funktionsumfang schon in älteren Versionen der Software enthalten. Seit Anfang 2019 vertreibt Siemens die Software NX nunmehr als „Continuous Release“ Version (momentan in Version 1872[3]), im gleichen Zuge wurde aber auch die MacOS Unterstützung eingestellt und die Linux-Variante nur noch ohne Graphical User Interface (GUI) als NX-Batch bereit gestellt. Im Rahmen dieser Arbeit wird jedoch nur die Windows Version und davon auch nur die CAD Funktionalität genutzt.

Das Siemens NX Softwarepaket umfasst zudem eine Sammlung von Application Programming Interfaces (APIs) namens **NX Open**, die für die Entwicklung von Erweiterungen für NX gedacht sind. Es werden mehrere etablierte Programmiersprachen, darunter C#, C/C++, Visual Basic, Java und Python (»Siemens Documentation: Programming Tools«, 2014) von NX Open unterstützt. Dabei kann das neue Programm in die Benutzeroberfläche (GUI) eingebunden oder als eigenständiges Programm aus der NX Hauptinstanz heraus aufgerufen werden.

Nachdem das Graphenersetzungssystem GrGen.NET nur eine Schnittstelle für C# aufweist, wird in dieser Arbeit dementsprechend NX Open für C# verwendet. In *Visual Studio* wird dafür ein eigenes Projekt erstellt und zusätzlich zu den GrGen Bibliotheken (vgl. Abschnitt 3.1) die entsprechenden NX Open Bibliotheken eingebunden. Zum Debuggen muss außerdem eingestellt werden, dass eine Instanz von Siemens NX gestartet wird und daran der remote Debugger angehängt wird. In der gestarteten NX Instanz muss dann über die Menüoption „`File -> Execute -> NX Open...`“ das kompilierte Programm ausgeführt werden.

Die mitgelieferten NX Open Bibliotheken, `NXOpen.dll`, `NXOpen.Guide.dll`, `NXOpen.UF.dll`, `NXOpen.Utilities.dll`, `NXOpenUI.dll` liegen hierfür im Unterverzeichnis `NXBIN/managed` des Installationspfades vor. In dieser Arbeit ist aber nur die Erste der genannten Dateien (`NXOpen.dll`) von Bedeutung, mit ihr ist der grundlegende Funktionsumfang von NX verfügbar.

Die **Guide Bibliothek** liefert zusätzliche simplifizierte Funktionen, die geometrische Objekte erstellen und alternativ zu den Standardfunktionen eingesetzt werden können (vgl. Siemens, 2016, S. 133). Aufgrund ihrer limitierten Einsatzmöglichkeiten und zur Wahrung eines einheitlichen Programmierstils finden diese Funktionen in dieser Arbeit jedoch keine weitere Anwendung.

In der **UF Bibliothek** sind einige Wrapper zu mittlerweile veralteten NX Open C Funktionen enthalten, die man in einer alten NX Version als *User Functions* bezeichnet hat (vgl. Siemens, 2016, S. 43). Bestimmte Teilaufgaben in dieser Arbeit können zwar mit diesen Funktionen gelöst werden, es existieren aber für alle hier betrachteten Fälle modernere und besser geeignete Funktionen wodurch diese Bibliothek auch nicht weiter von Bedeutung ist.

[2] Release date: 27. Oktober 2017
[3] Release date: 26. Juni 2019

Die **Utilities Bibliothek** liefert einige Hilfsfunktionen und -klassen, unter anderem den **NXObjectManager**. Dieser Manager beinhaltet ein Mapping zwischen NX Objekten und deren Tags, d.h. dessen Hilfe können NX Objekte über den entsprechenden Tag angesprochen werden. In dieser Arbeit werden aber nur sehr kleine Modellbeispiele behandelt, daher können die NX Objekte direkt in entsprechenden Klassenattributen gespeichert werden und direkt angesprochen werden. Somit findet auch diese Bibliothek hier keine Anwendung.

Zuletzt ist noch die **NXOpenUI Bibliothek** zu nennen. Sie beinhaltet Funktionen, die die GUI betreffen und kommt nur zum Einsatz, wenn ein Programm geschrieben werden soll, das über die Benutzeroberfläche von NX gesteuert werden kann. In dieser Arbeit wird jedoch ein schlichtes Übersetzungstool entwickelt, das keine weitere Steuerung benötigt, eine sogenannte *Batch Application*. Daher ist auch diese Bibliothek von keinem weiterem Interesse.

Einen grundlegenden Einblick in die Funktionalität von NX Open liefert ein **Beginner's Guide** (vgl. Siemens, 2016). Diese Anleitung ist jedoch für die Programmiersprache Visual Basic geschrieben und führt nur an einfache Problemstellungen heran. Auch ein **Reference Guide**, der eine Auflistung aller NX Open Klassen und Funktionen darstellt, ist vorhanden und über die Online-Dokumentation beziehbar (»Siemens Documentation: Programming Tools«, 2014). Beide Guides sowie die Online Dokumentation sind für den Einstieg in die NX Open Benutzung sehr hilfreich, kommen aber bei komplexeren Problemstellungen schnell an ihre Grenzen. Daher ist an dieser Stelle das Journaling Tool von NX zu erwähnen.

Das **Journaling Tool** ermöglicht dem Anwender eine NX Sitzung aufzuzeichnen als entsprechenden API-Code. Die Aufzeichnung lässt sich zu jedem Zeitpunkt einer Sitzung über „`Menü -> Tools -> Journal -> Record...`“ starten und kann in allen unterstützten Programmiersprachen ausgegeben werden. Im GUI werden dann alle aufzeichenbaren Tools und Einstellungen mit einem grünen Quadrat markiert und es wird entsprechend Code generiert, wenn diese angewendet werden. Der generierte Code umfasst dabei aber auch Hintergrundprozesse, wie das Setzen von Rückgängig Markern oder das neu setzen von voreingestellten Parametern, und kann auch bei einfachen Aufgaben schnell mehrere Hundert Zeilen lang werden. Jedoch können auf diese Weise die nötigen Funktionen und Einstellungen für bestimmte Prozesse herausgefunden werden und für das eigene Projekt interpretiert werden.

3.3 FreeCAD

Die Software FreeCAD ist ein frei erhältliches, parametrisches CAD System, das seit 2001 auf Basis des geometrischen Modellierungskerns *Open Cascade* entwickelt wird. Open Cascade, vormals CAS.CADE (eine Abkürzung für Computer Aided Software for Computer Aided Design and Engineering), wurde erst im Vorjahr unter einer Open Source Lizenz veröffentlicht und stellt einen Satz an Bibliotheken dar, die die Grundlage für die 3D Modellierung in

FreeCAD darstellen. Diese und alle in diesem Abschnitt folgenden Informationen zu FreeCAD beruhen auf der Online Dokumentation von FreeCAD (»Über FreeCAD«, 2019).

FreeCAD liegt aktuell in der Version 0.18[4] vor und unterstützt mittlerweile eine große Bandbreite an Modellierungswerkzeugen, so dass die Software durchaus mit seinen kommerziellen Gegenstücken, wie z.B. Catia oder sogar Solid Edge bzw. NX, verglichen werden kann. Außerdem ist FreeCAD vollständig plattformübergreifend und kann daher mit demselben Funktionsumfang auf Linux, MacOS und Windows genutzt werden. Das CAD System ist dabei modular aufgebaut, d.h. es besteht aus dem Grundprogramm, aus dem in verschiedene Module gewechselt werden kann, die jeweils eine unterschiedliche Sammlung an Modellierungswerkzeugen für unterschiedliche Aufgabenbereiche aufweisen.

Eines der ersten verfügbaren Module war das **Part-Modul** mit dem zunächst nur in einfachen Constructive Solid Geometry (CSG) Abläufen 3D Geometrien über die Benutzeroberfläche erstellt werden konnten. Daraufhin kamen dann weitere Module hinzu, zunächst das **Draft-Modul**, mit dem 2D-Zeichnungen angefertigt werden können und später – ab Version 0.10[5] – unter anderen auch das **Sketcher-Modul**, zur Erstellung parametrischer Skizzen, die dann mit dem Part-Modul zu 3D Körpern extrudiert werden können. Ein mit der offiziellen Version veröffentlichtes Assembly-Modul, mit dem Baugruppen erstellt und bearbeitet werden können, gibt es derzeit jedoch noch nicht. Allerdings gibt es Assembly Module, die von der FreeCAD-Community entwickelt wurden, aber deswegen momentan noch nicht gut dokumentiert und wenig leistungsstark sind.

FreeCAD wird hauptsächlich in der Programmiersprache C++ entwickelt, es wurde aber zusätzlich ein Python Interface integriert, mit dem auf den vollen Funktionsumfang von FreeCAD zugegriffen werden kann. Im Hauptfenster von FreeCAD wird standardmäßig eine Python Konsole mit angezeigt, auf der alle vom Benutzer durchgeführten Aktionen als Python-Code ausgegeben werden, ganz ähnlich zu dem Journaling Tool von Siemens NX (vgl. Abschnitt 3.2). Außerdem können hier alle Funktionen von FreeCAD direkt als Python-Code eingegeben werden.

Es können auch externe Python Skripte geschrieben werden, die mit der Python API den gesamten Funktionsumfang von FreeCAD nutzen können. Für diese Arbeit wird daher mit der IDE *pyCharm* ein Python Projekt angelegt und das dazu nötige und von FreeCAD mitgelieferte Python Modul `FreeCAD.pyd` importiert. Die einzelnen gewünschten FreeCAD Module (Sketcher und Part) werden über separate Python Module geladen. Und da das Graphenersetzungssystem GrGen.NET ausschließlich eine Schnittstelle für C# besitzt, ist es zudem nötig für dieses Projekt eine Übersetzungs-Bibliothek für dessen Gebrauch mit Python zu entwickeln. Ergänzend ist noch zu erwähnen, dass wegen dem FreeCAD Modul für Python nur ein Interpreter der Version 2.7 verwendet werden kann.

[4] Release date: 12. März 2019
[5] Release date: 24. Juli 2010

Kapitel 4

Methodisches Vorgehen

Nachdem in den vorhergehenden Abschnitten die Problemstellung erläutert, auf die theoretischen Grundlagen eingegangen und die verwendete Software beschrieben wurde, wird im folgenden Kapitel nun detailliert die Herangehensweise zur Lösung der Problemstellung beschrieben. Zunächst wird hierzu der Aufbau der zu interpretierenden Graphen untersucht und die nötigen Schritte zur erfolgreichen Interpretation daraus abgeleitet. Schließlich wird die Entwicklungsarbeit der beiden Übersetzungstools für Siemens NX und FreeCAD getrennt aufgeschlüsselt und dabei aufgetretene Probleme besprochen.

Die beiden Tools sollen Graphen, die 3D Geometrien darstellen, einlesen und interpretieren können und schließlich in ein parametrisches Modell überführen. Für Siemens NX kann hierzu, wie in Abschnitt 3.2 beschrieben, entweder ein eigenständiges Programm (Kommandozeilenprogramm) oder ein in die GUI eingebettetes Programm (Dynamic Link Library (DLL)) entwickelt werden. Da aber während der Übersetzung keine Eingriffe durch den Benutzer erforderlich sein sollen, ist eine Einbettung in die GUI unnötig und es wird ein Kommandozeilenprogramm implementiert. Im Fall von FreeCAD wird das Tool in Form eines Python Skripts erstellt. Hierfür sind zwar auch Funktionen verfügbar, die einen Benutzereingriff zulassen, aber diese werden aus demselben Grund ausgespart.

4.1 Vorüberlegungen

Die in dieser Arbeit zu interpretierenden Graphen basieren auf einem Metamodell, dass dazu entworfen wurde parametrische Modelle genau abzubilden (vgl. Abschnitt 2.2.3). Jedoch ist bei der Überführung von der graphenbasierten in die parametrische Darstellung immer eine gewisse „Lesereihenfolge“ zu beachten – zum Beispiel kann keine Skizze erzeugt werden, wenn nicht zuvor eine Bauteil-Datei erstellt wurde. Daher ist es nötig zunächst anhand von Beispielgraphen eine allgemein gültige Vorgehensweise zu erarbeiten (siehe auch Abschnitt 2.1.3).

Die beiden Übersetzungstools werden daher anhand von einem sehr einfach aufgebauten Graphen entwickelt und mit einem weiteren etwas komplexeren Beispiel geprüft. Der erste einfache Graph stellt dabei zwei IPE-Träger dar (vgl. Abschnitt 2.1.3), d.h. er wird in eine einzelne Baugruppe mit zwei Bauteilen, die jeweils eine Skizze und eine entsprechende Extrusion beinhalten. Der zweite Graph bzw. das zweite Graphensystem stellt einen Tunnelabschnitt in verschiedenen Detailstufen dar. In der höchsten Detailstufe, die hierzu bearbeitet wird (LOD4), besteht das übersetzte parametrische Modell aus einer Baugruppe mit einem Bauteil, dass aber ein Alignment, sieben Skizzen, dazu sieben Sweeps und eine boolsche Operation enthält.

4.1.1 Programmstruktur

Betrachtet man nun zunächst die zu übersetzenden Graphen nur entlang der entsprechenden prozeduralen Kanten, kann man sehr schnell die in Abschnitt 2.1.3 besprochene Datenstruktur im Subgraphen wiedererkennen (vgl. Abb. 4.1 und 4.2). Die beiden hier dargestellten Subgraphen sind zur besseren Darstellbarkeit etwas vereinfacht indem alle Skizzenelemente ausgeblendet wurden. Zudem sind hier nur zwei einfache Graphen dargestellt – das IPE-Träger Beispiel und die Tunnelstrecke in Detailstufe LOD2 – da komplexere Graphen kaum übersichtlich abbildbar sind und prinzipiell keine größere Aussagekraft haben.

Allen hier beobachteten Teilgraphen dieser Art ist gemein, dass es einen Wurzelknoten gibt (Assembly oder Part) und jede prozedurale Kante eine *Parent-Child*-Beziehung zwischen den entsprechenden Knoten symbolisiert. Dabei ist der *Parent*-Teil am Ursprung der Kante und der *Child*-Teil am Ziel der Kante zu finden. Jeder Knoten in den Graphen, sei es ein prozeduraler oder Skizzen-Knoten, stellt zudem ein im CAD-System darstellbares Objekt dar, das je nach Knotentyp eine unterschiedliche Erstellungsmethode besitzt.

Es ist also sinnvoll, jeden Knotentyp des Graphensystems auch als eigene Klasse im Übersetzungstool zu implementieren. An diese Klassen werden zum einen alle Attribute der entsprechenden Knoten übertragen und zum anderen alle *Parents* und *Childs* in entsprechende neue Attribute gespeichert. Außerdem wird in jeder Klasse die Erstellungsmethode für das CAD-System als eigene Methode definiert und da alle Knoten vom Typ her sehr ähnlich sind, macht es Sinn alle Klassen von einer Basisklasse, hier `Entity`, zu vererben.

Die Instanzierung der einzelnen Klassen kann dann ausgehend vom Wurzelknoten stattfinden und entlang der prozeduralen Kanten fortgeführt werden. Auf diese Weise können alle Knoten ausgelesen und gleichzeitig alle *Parents* und *Childs* verknüpft werden. Einzelne Knoten können auf diese Weise nicht übersehen werden, da alle Knoten über mindestens einen Weg (Reihe an Kantenverbindungen) mit dem Wurzelknoten verbunden sein müssen – es kann kein geometrisches Element außerhalb einer Baugruppe oder eines Bauteils existieren.

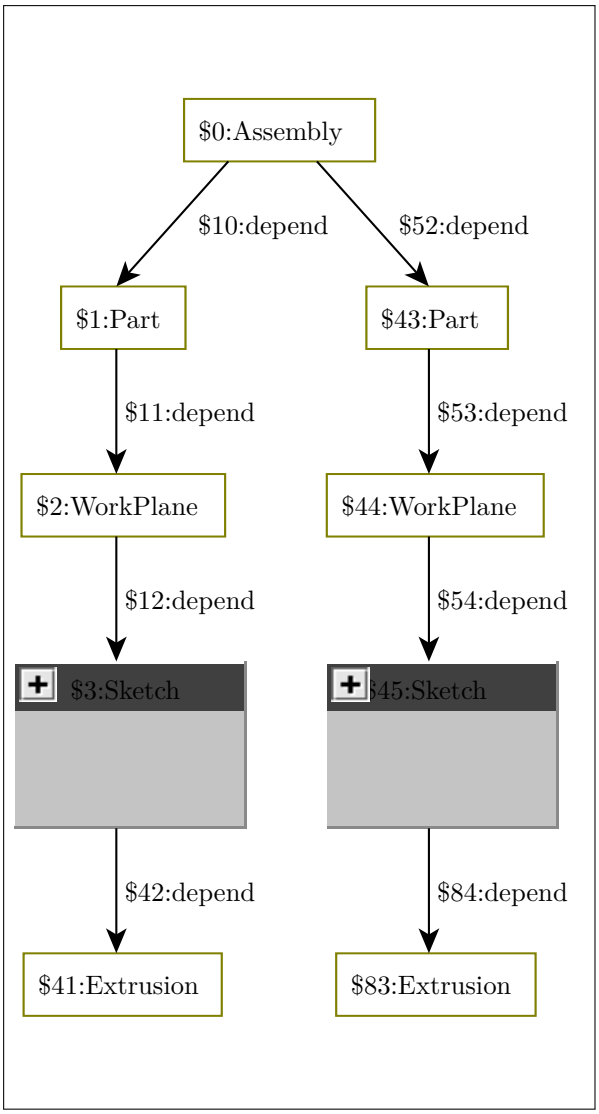


Abbildung 4.1: Subgraph der Graphendarstellung von zwei IPE-Trägern entlang seiner proceduralen Kanten (vgl. Beispiel aus Abschnitt 2.1.3).

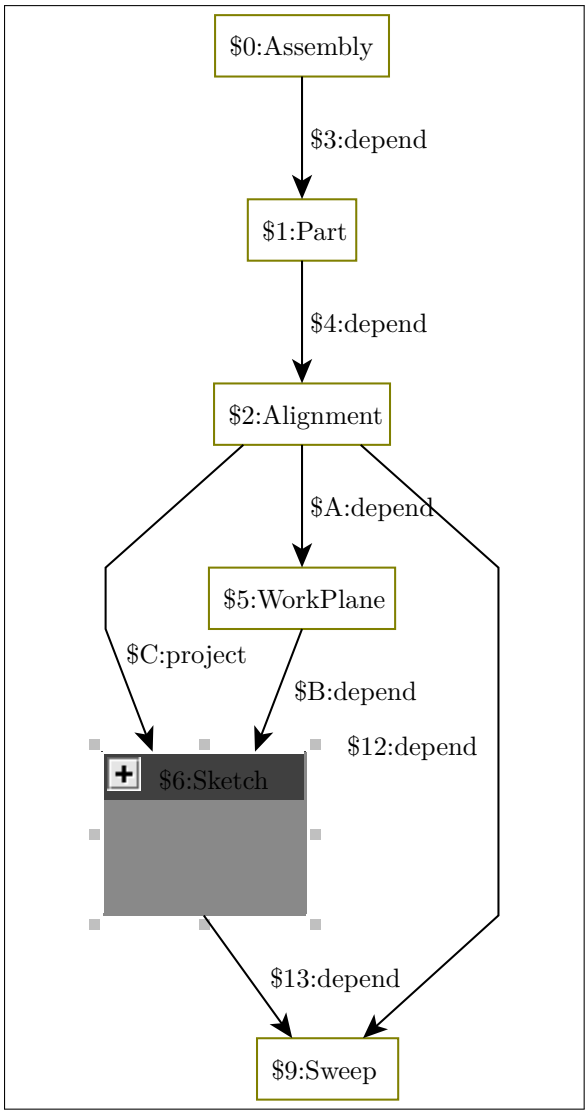


Abbildung 4.2: Subgraph der LoD 2 Graphendarstellung eines Tunnels entlang seiner prozeduralen Kanten.

Im Gegensatz zu den prozeduralen Kanten, die implizit als Parent und Child Attribut festgehalten werden, müssen Skizzen-Kanten als eigene Klassen repräsentiert werden. Sie repräsentieren im CAD-System Zwangsbeziehungen und müssen daher separat erstellt werden. Es werden also auch eigene Klassen für die verschiedenen Kantentypen implementiert, die von der Basisklasse `Constraint` abgeleitet werden. Auch hier werden alle im Metamodell definierten Attribute mit übergeben und sowohl Ursprungs- als auch Zielknoten als eigenes Attribut abgespeichert.

Wird der Teilgraph von nur einer Skizze betrachtet (vgl. Abb. 4.3), kann nur schwer eine höhere Struktur erkannt werden. Die Reihenfolge, in der die Constraint-Klassen instanziert werden ist

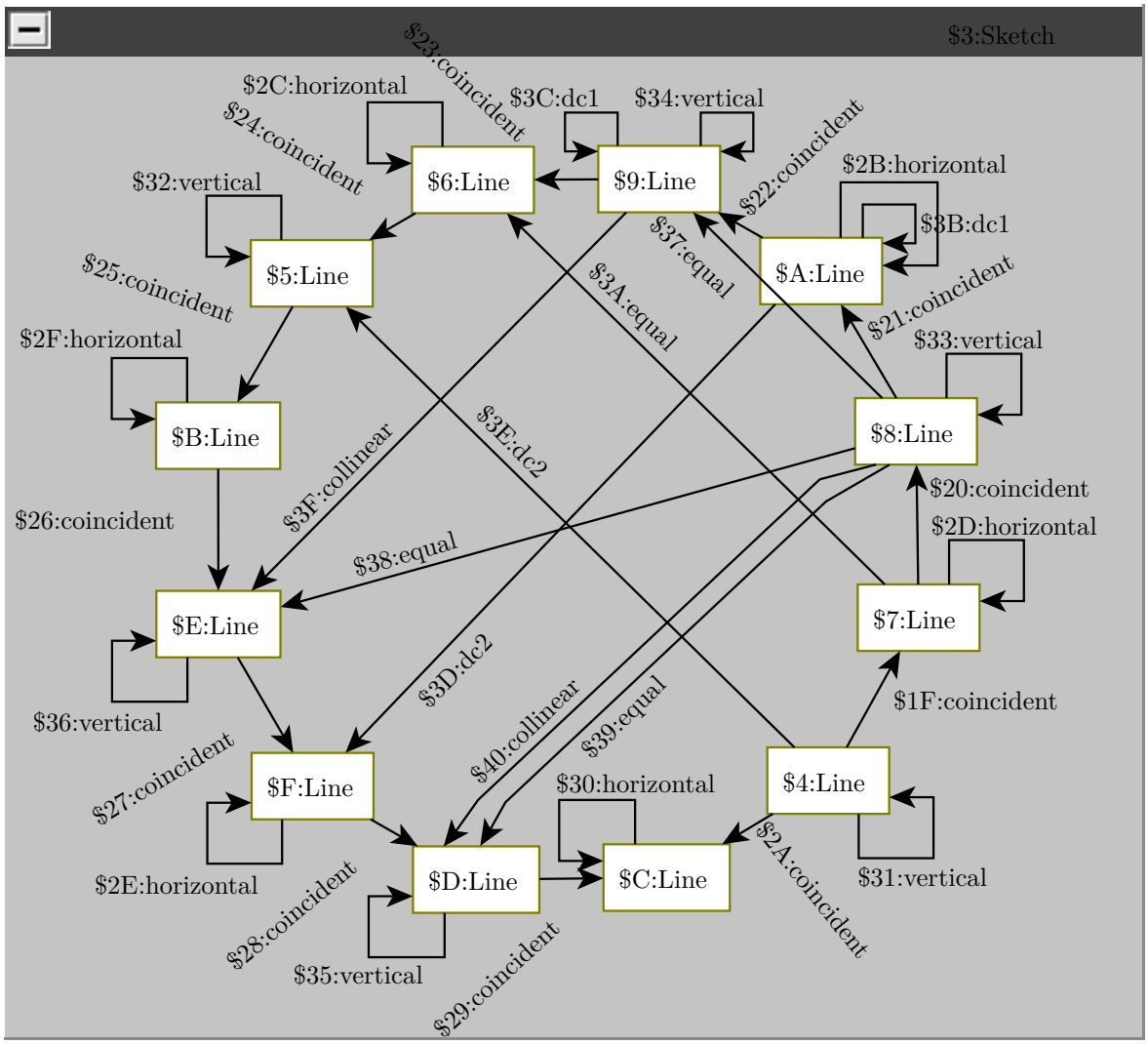


Abbildung 4.3: Vergrößerte Graphendarstellung eines Sketches aus Abb. 4.1.

hier zwar beliebig, es sollte allerdings darauf geachtet werden, dass zuvor alle Entity-Klassen instanziert wurden. Eine korrekte Verknüpfung als Ursprungs- oder Zielknoten ist dann leichter zu bewerkstelligen. Der Teilgraph in Abb. 4.3 ist absichtlich kreisförmig angeordnet, da man so leicht den Polygonschluss erkennen kann. Alle Kanten entlang des Kreisumfangs sind *coincident*-Kanten und verbinden die einzelnen Liniensegmente zu einem geschlossenen Polygon. Nur ein geschlossenes Polygon kann später in einen 3D-Körper extrudiert werden.

Gesteuert wird das entsprechende Programm über eine Hauptfunktion, in der zunächst das GrGen.NET Graphensystem geladen wird. Der zu übersetzende Graph wird dann über die API von GrGen direkt im Programm erstellt, kann alternativ aber auch über die `GXLImport.Import` Funktion der API direkt aus einer GXL-Datei gelesen werden. Der Graph wird – egal durch welche Methode erzeugt – in einer Instanz vom Typ `LGSPGraph` gespeichert. Die Knoten und

Kanten können dann über die Attribute `.Nodes` und `.Edges` ausgelesen und weiterverarbeitet werden.

4.1.2 Erzeugung der Geometrien im CAD-System

Wie zuvor erwähnt muss bei der Erstellung der Geometrien eines parametrischen Modells auf eine gewisse Reihenfolge achtgegeben werden. Da aber in den entsprechenden Entity-Klassen Parents und Children verlinkt werden, ist dies sehr einfach zu bewerkstelligen. Hierzu wird vor der Erzeugung einer Geometrie geprüft ob auch die Parent-Geometrien bereits erstellt wurden. Fehlen diese Parent-Geometrien werden diese zuerst erstellt. Für diese Existenz-Prüfung muss damit den Entity-Klassen ein weiteres Attribut zugewiesen werden, ein boolscher Wert, der anzeigt ob die Geometrie bereits existiert.

Bei der Erstellung der einzelnen Sketch-Elemente, also der Linienelemente, ist keine Reihenfolge zwingend einzuhalten, auch die Abfolge in der die Zwangsbedingungen an diese Elemente angebracht werden ist prinzipiell willkürlich. Jedoch löst der GCS die Gesamtgeometrie beim Anbringen jeder einzelnen Zwangsbedingung neu und muss eventuell einige Elemente verschieben oder verdrehen. Wenn die entsprechenden Geometrien nicht wie in Abschnitt 2.1.4 erläutert sinnvoll positioniert sind, kann es zu unerwünschten Verschiebungen und so zu ungeplanten Ergebnissen kommen. Im schlimmsten Falle sind solche Geometrien dann auch nicht mehr lösbar, wenn weitere Zwangsbedingungen angebracht werden.

Ungünstige Verschiebungen können zwar durch eine geschickte Abfolge beim Anbringen der Einschränkungen umgangen werden, aber es kann keine solche sinnvolle allgemeine Vorgehensweise für das Anbringen der Zwangsbedingungen einfach aus dem Graphen abgeleitet werden, zumal es auch für jeden Graphen mehr als eine „richtige“ Herangehensweise gibt. Es ist daher auch unmöglich hierzu ein adaptives System zu implementieren, welches sicher stellt, dass alle Geometrien immer richtig verschoben werden. Wie gut die Qualität der Übersetzung ist hängt also direkt davon ab, wie sorgfältig der zu interpretierende Graph erstellt wurde.

Verschiedene CAD-Systeme liefern zudem üblicherweise verschiedene GCS mit, die mehr oder weniger robust sein können, d.h. manche Solver sind deutlich weniger fehleranfällig. Laut der Erfahrungen, die bei der Bearbeitung dieser Thesis gemacht wurden, ist z.B. der GCS von NX deutlich stabiler als der von FreeCAD. Im Falle von FreeCAD müssen daher bei manchen Problemen Hilfsconstraints eingesetzt werden, wie z.B. das temporäre Blockieren einer Linie um dessen Verschiebung zu vermeiden.

4.2 G2NX

Siemens bietet einen leichten Einstieg in die Programmierung mit NXOpen indem zur Software NX auch ein Visual Studio Projekt Template mitgeliefert wird, mit dem recht schnell ein neues NXOpen Projekt eingerichtet werden kann. Zur Orientierung gibt es im NX-Unterordner `UGOPEN` zudem einige Programmbeispiele für alle unterstützten Programmiersprachen. Diese Beispiele erscheinen jedoch sehr veraltet, da verstärkt die in Abschnitt 3.2 erwähnten *User Functions* verwendet werden. Nichtsdestotrotz wurden aufbauend auf ersten Tests mit der Beispielapplikation `EX_Curve_CreateArc.cs` erste Erfahrungen gesammelt. Von dem CreateArc-Beispiel wurde z.B. die grundsätzliche Struktur der Hauptfunktion (`int Main()`) und die Implementierung eines *Stream Writers* zur Log-Datei Ausgabe abgeschaut.

4.2.1 Grundstruktur des Programms

Beim Starten der Hauptfunktion wird gleich eine `StreamWriter` Instanz geladen und damit eine Log-Datei erstellt. Daher wird jede kritische Befehlskette in einen *try-catch*-Block eingebettet und eventuell auftretende Fehler in diese Log-Datei ausgegeben. Genauso wird nach jedem erfolgreichen Abschluss wichtiger Codesegmente eine bestätigende Log-Ausgabe erzeugt. Beim Beenden des Programms wird dann zusätzlich noch eine Zusammenfassung der erstellten Objekte in die Log-Datei geschrieben.

In der Hauptfunktion soll dann zunächst der Graph eingelesen werden (vgl. Vorüberlegungen in Abschnitt 4.1.1). Dies geschieht im Laufe der Entwicklung direkt im Code, d.h. über die GrGen.NET-API wird der entsprechende Graph direkt erstellt. Das hat zum Vorteil, dass bei kleinen Änderungen in den GrGen.NET Dateien nicht jedes Mal die Graphen neu exportiert werden müssen. Später kann dies aber mit einer Import-Funktion ersetzt werden.

Die Knoten und Kanten sollen dann in entsprechende Objekte umgewandelt und über deren `CreateNXGeometry()`- bzw. `CreateNXConstraint()`-Methode in NX gezeichnet werden. Um das zu bewerkstelligen müssen also zuerst für die verschiedenen Typen von Knoten und Kanten Klassen definiert werden – diese werden im Folgenden ganz analog zum Metamodell (vgl. Abschnitt 2.2.3) des Graphensystems angelegt. Danach muss ein System entwickelt werden, das in sinnvoller Abfolge analog zum Graphen die entsprechenden Objekte instanziert und miteinander verknüpft und schließlich die Erstellungsmethoden aufruft.

Zum Ende der Hauptfunktion wird die aktive NX-Session gespeichert und eine Zusammenfassung der erstellten Objekte in die Log-Datei ausgeschrieben. Nach dem Beenden des Übersetzungstools ist die Sitzung in NX weiterhin geöffnet und die Ergebnisse können evaluiert und nach Belieben nachbearbeitet werden.

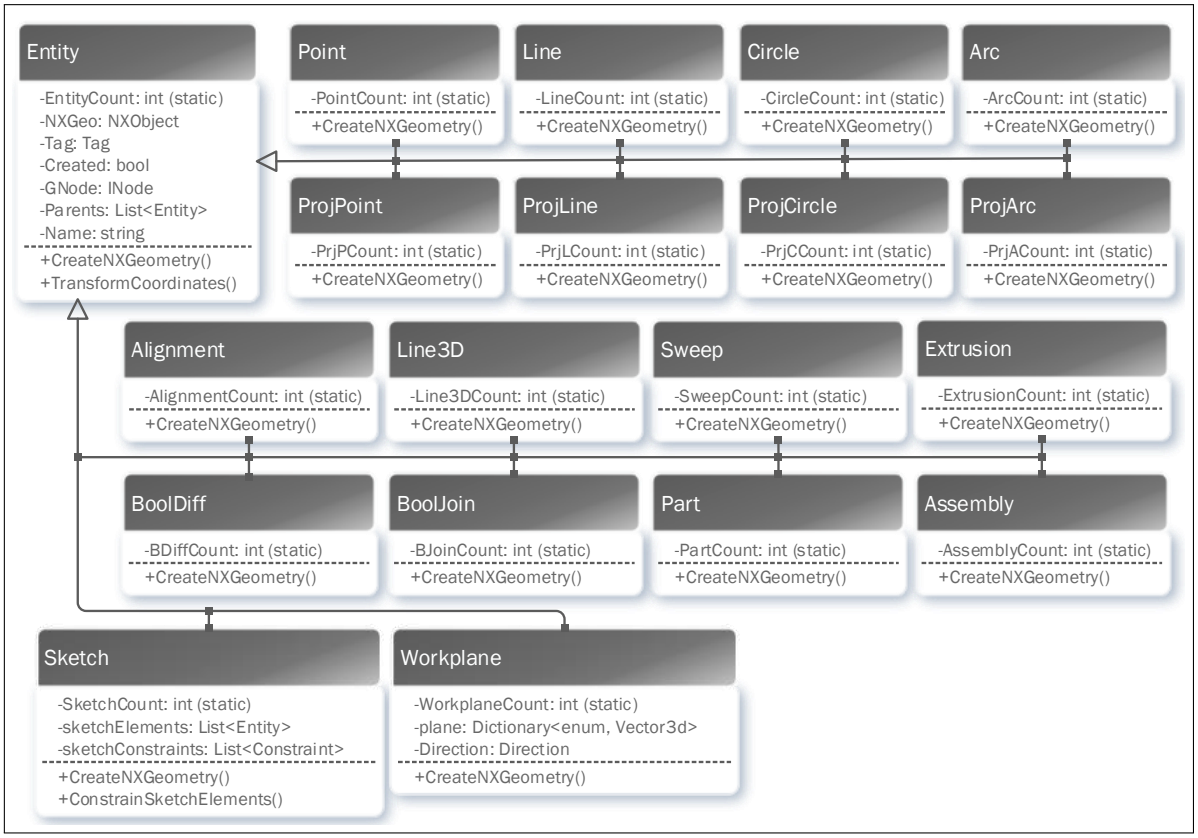


Abbildung 4.4: Klassenstruktur der Knoten. Die Unterklassen, die von rechts anschließen, stellen die SketchNodes dar und die, die von unten anschließen, stellen die ProceduralNodes dar, hierbei werden nur die wichtigsten Attribute und Methoden gezeigt.

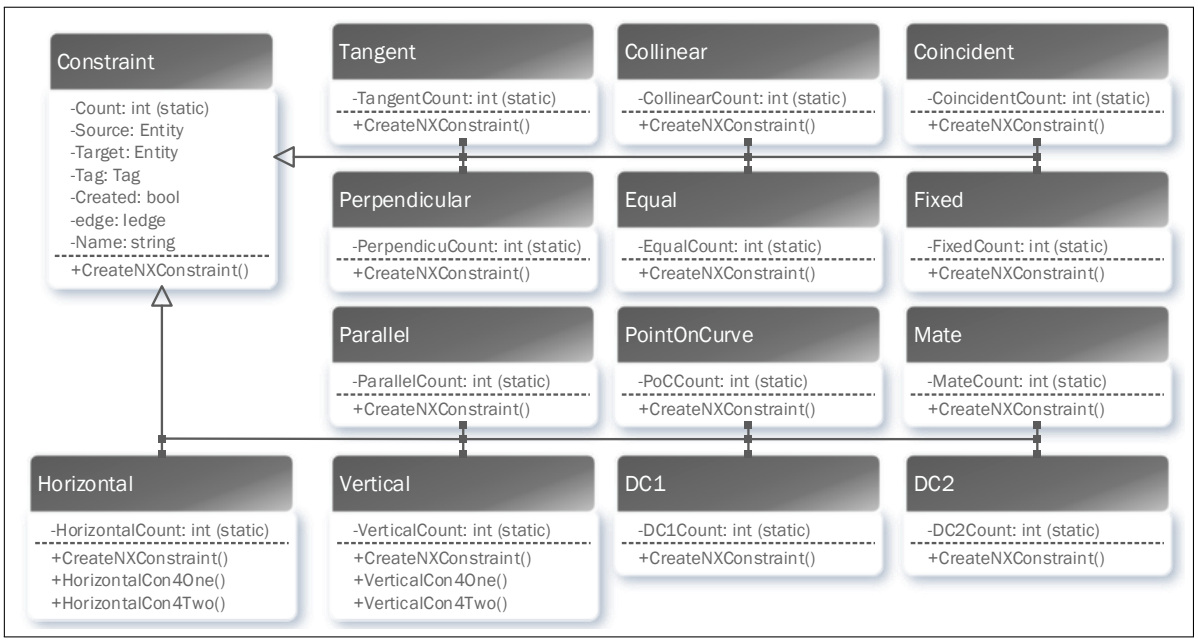


Abbildung 4.5: Klassenstruktur der Kanten. Auch hier sind nur die wichtigsten Attribute und Methoden genannt.

Die implementierte Klassenstruktur der Knoten wird in Abb. 4.4 dargestellt, dabei sind nur die wichtigsten Attribute und Methoden aufgeführt. Zu beachten ist, dass die Basisklasse `Entity` eine abstrakte Klasse ist und daher nicht instanziert werden kann. Die Methode `CreateNXGeometry()` der Basisklasse `Entity` wird in den Unterklassen jeweils überschrieben. Zudem ist für jede Unterklasse ein eigener Counter implementiert mit dem unter anderem einzigartige Namen generiert werden können.

Abbildung 4.5 zeigt in gleicher Weise die Klassenstruktur der Kanten. Auch hier ist die Basisklasse abstrakt, die Unterklassen mit einem eigenen Counter versehen und wird die Methode `CreateNXConstraint()` in jeder Unterklasse überschrieben. Die beiden Constraints *horizontal* und *vertical* können hier entweder auf ein Element angewendet werden, im Graph repräsentiert durch eine Schlinge, oder auf zwei Elemente. Im letzteren Fall ist die Ausrichtung zueinander gemeint, z.B. zwei Punkte die vertikal zueinander sind, liegen auf derselben gedachten vertikalen Linie.

4.2.2 Ablauf der Übersetzung

Für einfache Graphen, wie z.B. dem IPE-Träger Beispiel, kann eine feste Reihenfolge, in der die einzelnen Erstellungsmethoden aufgerufen werden sollen, festgelegt werden. Im Falle des IPE-Trägers muss demnach folgende Reihenfolge ausgeführt werden:

1. Erstelle Baugruppe.
2. Erstelle Part und füge es der Baugruppe hinzu.
3. Lege Arbeitsebene fest.
4. Kreiere Skizze in der zuvor festgelegten Arbeitsebene.
5. Zeichne alle Sketch-Elemente.
6. Wende nacheinander alle Zwangsbedingungen an.

Diese Reihenfolge funktioniert **nur genau dann**, wenn jedes prozedurale Element nur einen Parent besitzt und die vorgegebene Struktur daher nicht weiter verzweigt ist. Werden zudem z.B. Sweeps verwendet schiebt sich auch noch die Erstellung des Alignments zwischen Schritt 2 und 3. Es muss daher ein adaptiveres System implementiert werden.

Im Folgenden wird aus diesem Grund eine *RelationBuilder*-Klasse implementiert (vgl. Abb. 4.6). In einer Instanz dieser Klasse können dann alle Informationen gesammelt und geordnet ausgeführt werden. Als eigene Klasse ist diese Codestruktur sehr modular und damit bei Bedarf schnell anpassbar. Dem Constructor des RelationBuilders wird bei der Initialisierung lediglich eine Liste der Knoten und Kanten aus dem Graphen übergeben und dann in einer festen Reihenfolge die eigenen Methoden aufgerufen (vgl. Algorithmus 4.1).

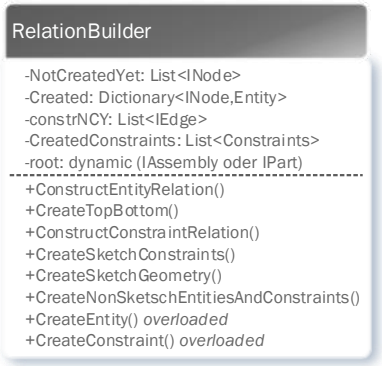


Abbildung 4.6: RelationBuilder-Klasse.

Algorithmus 4.1: Instanzierung der RelationBuilder-Klasse.

```
RelationBuilder rBuilder;
rBuilder = new RelationBuilder(WorkingGraph.Nodes.ToList(),
                               WorkingGraph.Edges.ToList());
rBuilder.ConstructEntityRelation();
rBuilder.ConstructConstraintRelation();
rBuilder.CreateSketchGeometry();
rBuilder.CreateSketchConstraints();
rBuilder.CreateNonSketchEntitiesAndConstraints();
```

Constructor

Bei der Instanzierung des RelationBuilders werden zwei Listen übergeben, zum einen die Knoten als `INode` Datentyp und zum anderen die Kanten als `IEdge` Datentyp. Intern werden diese beiden Listen in den Attributen `NotCreatedYet` und `ConstrNCY` temporär gespeichert. Gleich anschließend werden aus der Kantenliste alle prozeduralen Kanten entfernt, da diese wie zuvor gesagt nur implizit über die Parent-Child-Beziehungen der Klassen gespeichert werden. Zuletzt wird noch der Wurzelknoten identifiziert und in `root` gespeichert.

Algorithmus 4.2: Constructor der RelationBuilder-Klasse.

```
public RelationBuilder(List<INode> ncy, List<IEdge> cNCY)
{
    NotCreatedYet = ncy;
    constrNCY = cNCY;
    constrNCY.RemoveAll(e => e is IproceduralEdge);
    root = NotCreatedYet.Find(n => n.Incoming.Count() == 0);
}
```

ConstructEntityRelation()

Diese Methode übernimmt die Instanzierung der einzelnen Entity-Klassen entsprechend der Knoten aus dem zu übersetzenden Graphen. Hierbei wird wie zuvor in Abschnitt 4.1.1 überlegt vom Wurzelknoten ausgehend entlang der prozeduralen Kanten für jeden Knoten die entsprechende Klasse instanziert. Zu diesem Zweck ruft die `ConstructEntityRelation`-Methode die Rekursive Methode `CreateTopBottom()` mit den Parametern `root` und `null` auf (vgl. Algorithmus 4.3). Ist die entsprechende Klasse schon instanziert wird nur das mit übergebene Parent Objekt im Attribut `.Parents` gespeichert.

Innerhalb dieser Methode wird zunächst geprüft ob eine Klasse schon instanziert wurde und wenn nicht über die überladene Methode `CreateEntity()` instanziert sowie zur späteren Abrufbarkeit in dem Dictionary `Created` gespeichert. Gleichzeitig wird das mit übergebene Parent Objekt verlinkt.

Anschließend wird für jede prozedurale Kante jeweils erneut die Methode `CreateTopBottom()` aufgerufen mit dem Ziel der Kante und dem gerade erzeugten Objekt als Parameter parent. Nur im Falle einer *contain*-Kante werden auch die Childs in dem zuvor erstellten Objekt gespeichert (ausschließlich bei Skizzen notwendig).

Algorithmus 4.3: Instanzierung der Entity-Klassen entsprechend der Knoten im Graphen.

```
private void CreateTopBottom(dynamic node, dynamic parent)
{
    if (NotCreatedYet.Contains(node))
    {
        Created.Add(node, CreateEntity(node));
        NotCreatedYet.Remove(node);
        Created[node].SetParent(parent);

        foreach (IEdge procedural in
                 ((INode)node).Outgoing.OfType<IproceduralEdge>())
        {
            CreateTopBottom(procedural.Target, Created[node]);
            if (procedural is Icontain)
            {
                Created[node].SetChild(Created[procedural.Target]);
            }
        }
    }
    else
    {
        Created[node].SetParent(parent);
    }
}
```

ConstructConstraintRelation()

Nachdem alle Entity-Objekte erstellt sind, kann mit dieser Methode die Instanzierung der Constraint-Klassen durchgeführt werden. Dafür wird für jede Kante in der temporären Liste `constrNCY` ein *Constraint*-Objekt instanziert mithilfe der überladenen `CreateConstraint()` Methode. Dieser werden die Kante selbst sowie das entsprechende Objekt für den Ursprungs- und Zielknoten als Parameter übergeben. Ist die Kante vom Typ sketchEdge, so wird das soeben erstellte Constraint-Objekt mit dem entsprechenden Sketch-Objekt verknüpft (Sketch Attribut `sketchConstraints`).

Algorithmus 4.4: Instanzierung der Constraints-Klassen entsprechend der Kanten im Graphen.

```
while (constrNCY.Count > 0)
{
    dynamic e = constrNCY.First();
    Constraints constr = CreateConstraint(e, Created[e.Source],
                                           Created[e.Target]);
    CreatedConstraints.Add(constr);
    if (e is Isketchedge)
    {
        INode sk = ((IEdge)e).Source.Incoming.
                   OfType<Icontain>().First().Source;
        ((Sketch)Created[sk]).SetSketchConstraint(constr);
    }
    constrNCY.Remove(e);
}
```

CreateSketchGeometry()

Nachdem sowohl Entity- als auch Constraints-Klassen instanziert sind, kann mit dem Zeichnen der Geometrien begonnen werden. Hierbei kann wie in Abschnitt 4.1.2 schon angesprochen mit einer zufälligen Klasse angefangen werden, da über das Parents-Attribut geprüft werden kann ob Abhängigkeiten eingehalten werden. Es hat sich jedoch als geschickt herausgestellt, das Zeichnen auf Ebene der Skizzen zu beginnen, d.h. in `CreateSketchGeometry()` wird nacheinander für jede Skizze die `CreateNXGeometry()` Methode aufgerufen, sofern die Skizze nicht schon erstellt wurde.

In allen Erstellungsmethoden der Entity-Klassen, also auch bei der Version der Sketch-Klasse, wird zunächst die Basismethode der Entity-Oberklasse aufgerufen. In diesem für alle Unterklassen gültigen Code wird die Existenz der Parents geprüft und gegebenenfalls deren Erstellungsmethode aufgerufen. Im Falle von Skizzen gibt es aber noch eine weitere Form von Abhängigkeit, nämlich genau dann, wenn in der Skizze ein projiziertes geometrisches Element

verwendet wird. Daher wird bei Skizzen als nächstes geprüft ob ein projiziertes Element vorhanden wird und wenn ja, wird zunächst die Skizze erstellt, aus der projiziert wird.

Sind alle Abhängigkeiten erfüllt, wird zunächst die Skizze über den `SketchInPlaceBuilder` von NXOpen erstellt und anschließend alle der Skizze zugehörigen geometrischen Elemente gezeichnet. An sich können alle Objekte entsprechend der Sketch-Knoten separat erstellt werden, aber da hierzu jedes Mal die entsprechende Skizze aktiviert und danach deaktiviert werden muss, ist es sinnvoller und zeitsparender dies für jede Skizze gebündelt durchzuführen.

An dieser Stelle ist die Programmierung mit NX Open etwas eigenartig, beim Erstellen von Skizzenobjekten müssen zuerst die Linienobjekte im 3D-Raum des Parts erstellt und erst dann der Skizze hinzugefügt werden. Im Graphensystem wird aber davon ausgegangen, dass die Linienobjekte im lokalen Koordinatensystem der Skizze gezeichnet werden, daher sind dort nur 2D-Koordinaten gespeichert. Aus diesem Grund vererbt die Klasse Entity die Methode `TransformCoordinates()` (vgl. Algorithmus 4.5). Übergibt man die x- und y-Skizzen-Koordinate eines Punktes an diese Methode, so wird mithilfe der Orientierungsmatrix der Skizze die 3D-Position im globalen Koordinatensystem berechnet und als NX-Datentyp `Point3d` zurückgegeben.

Anhand von Algorithmus 4.6 wird exemplarisch gezeigt, wie ein Linienobjekt in einer Skizze gezeichnet wird, hier am Beispiel einer Linie. Zu beachten ist hier, dass in der if-Verzweigung geprüft wird ob temporäre Koordinaten gegeben wurden. Wenn im Graphen keine temporären Koordinaten abgelegt wurden, so sind alle x- und y-Werte zu null gesetzt. Da aber eine Linie nicht zwischen zwei aufeinander fallenden Punkten erstellt werden kann muss einer der beiden Punkte verschoben werden. Ähnlich wird das auch bei den anderen Linienobjekten gehandhabt.

Zunächst werden also die Koordinaten transformiert und mit den beiden daraus resultierenden `Point3d` das Linienobjekt erstellt. Zur späteren Abrufbarkeit wird das so erhaltene `NXObjekt` in dem Attribut `NXGeo` abgespeichert. Dann wird der boolsche Wert `GeometryCreated` auf *true* gesetzt damit geprüft werden kann ob das NXObjekt schon erstellt wurde. Zuletzt wird

Algorithmus 4.5: Methode zur Transformation von lokalen Skizzen Koordinaten in globale Koordinaten.

```
protected Point3d TransformCoordinates(double x, double y, NXMatrix LCS)
{
    double xNew = x * LCS.Element.Xx + y * LCS.Element.Xy;
    double yNew = x * LCS.Element.Yx + y * LCS.Element.Yy;
    double zNew = x * LCS.Element.Zx + y * LCS.Element.Zy;
    var p = new Point3d(xNew, yNew, zNew);

    return p;
}
```

Algorithmus 4.6: Ausschnitt aus der Erstellungsmethode der Klasse Line.

```
Sketch sk = Parents.Find(de => de is Sketch) as Sketch;
NXOpen.Part workpart = sk.NXGeo.OwningPart as NXOpen.Part;
NXOpen.NXMatrix LCS = ((NXOpen.Sketch)sk.NXGeo).Orientation;

NXOpen.Point3d startP = TransformCoordinates(GNode.temp_Sx,
                                             GNode.temp_Sy, LCS);
NXOpen.Point3d endP;
if ( GNode.temp_Sx == GNode.temp_Ex && GNode.temp_Sy == GNode.temp_Ey )
{
    endP=TransformCoordinates(GNode.temp_Ex + 1, GNode.temp_Ey + 1, LCS);
}
else
{
    endP=TransformCoordinates(GNode.temp_Ex, GNode.temp_Ey, LCS);
}

NXGeo = workpart.Curves.CreateLine(startP, endP);
Tag = NXGeo.Tag;
GeometryCreated = true;
((NXOpen.Sketch)sk.NXGeo).AddGeometry(NXGeo as NXOpen.Line,
    NXOpen.Sketch.InferConstraintsOption.InferNoConstraints);
```

noch das Linienobjekt mithilfe der `.AddGeometry()` Methode der entsprechenden Skizze hinzugefügt. Hier ist der Übergabewert *InferNoConstraints* essenziell, da sonst von NX automatisch „sinnvolle“ Zwangsbedingungen angebracht werden, die später beim Anbringen der im Graphensystem definierten Bedingungen zu einer Überbestimmung der Skizze führen.

CreateSketchConstraints()

Sobald alle Geometrien gezeichnet sind ist es möglich die entsprechenden Zwangsbedingungen anzubringen. Auch hier ist es im Falle von Skizzeneinschränkungen sinnvoll diese immer für jede Skizze gebündelt anzubringen, da auch hier die entsprechende Skizze aktiviert werden muss. Die Methode `CreateSketchConstraints()` ruft daher nacheinander für jede Skizze die Methode `ConstrainSketchElements()` auf (vgl. Algorithmus 4.7).

Die Erstellungsmethoden können dabei so einfach sein wie nur eine NXOpen-Funktion aufzurufen oder auch deutlich komplexer in Form eines ConstraintBuilders, je nachdem wie viele Informationen für die Einschränkung benötigt werden. Kompliziert sind z.B. die dimensionalen Bedingungen, da sie unterschiedlich ausgeführt werden müssen, je nachdem auf welche Art und auf welche Geometrien sie angewendet werden sollen.

Algorithmus 4.7: Erzeugung der Zwangsbeziehungen innerhalb eines Sketches.

```
public void ConstrainSketchElements()
{
    NXOpen.Part workpart = NXGeo.OwningPart as NXOpen.Part;
    NXOpen.PartLoadStatus loadstat;
    Program.TheSession.Parts.SetDisplay(workpart, false, false, out
        loadstat);
    Program.TheSession.Parts.SetWork(workpart);
    ((NXOpen.Sketch)NXGeo).Activate(NXOpen.Sketch.ViewReorient.False);

    foreach (Constraints c in sketchConstraints)
    {
        c.CreateNXConstraint();
    }

    ((NXOpen.Sketch)NXGeo).Deactivate(NXOpen.Sketch.ViewReorient.False,
        NXOpen.Sketch.UpdateLevel.Model);
}
```

CreateNonSketchEntitiesAndConstraints()

Nachdem nun, sofern keine Fehler aufgetreten sind, alle Skizzen gezeichnet und vollständig eingeschränkt sind, müssen noch alle restlichen Objekte erzeugt werden. Darunter fallen alle 3D-Operationen und 3D-Einschränkungen. Hierzu wird einfach für jedes Element des Entity-Dictionaries `Created` geprüft ob es erstellt wurde und sonst die Erstellungsmethode aufgerufen. Gleiches wird für die Constraint-Liste `CreatedConstraints` durchgeführt. Sind diese beiden Listen durchlaufen sollten alle Geometrien gezeichnet und eingeschränkt sein.

4.2.3 Probleme bei der Implementierung

Siemens NX stellt ein außerordentlich robustes und umfangreiches CAD-System dar, das alle nötigen Werkzeuge für die Entwicklung eines vollständig funktionsfähigen Übersetzungstools mitliefert. Ein Nachteil dieser Vielfältigkeit ist jedoch, dass es für einige Operationen verschiedene Möglichkeiten zur Umsetzung gibt, mit entsprechenden Vor- und Nachteilen.

Eine weitere Schwierigkeit stellt die dimensionale Bedingung `dc2` dar. Das Graphensystem liefert hier die Informationen, zwischen welchen Positionen die Bemaßung angebracht werden soll und mit welchem Wert. Um aber diese Bemaßung erfolgreich in Siemens NX umzusetzen muss noch die Maßmethode abgeleitet werden. Es kann hier aus den folgenden Methoden ausgesucht werden:

- Horizontal
- Vertical

- PointToPoint
- Perpendicular
- Cylindrical

Welche Methode gewählt werden muss hängt davon ab welche Geometrien zueinander bemaßt werden und welche Geometrie-Ports angegeben sind. Durch diese Vielfalt an Möglichkeiten wird die Erstellungsmethode sehr umfangreich mit mehreren If-else-Verzweigungen und überladenen Methoden.

4.3 G2FreeCAD

Die Programmierung mit der API von FreeCAD gestaltet sich sehr einfach und intuitiv, da FreeCAD einen eingebauten Python-Interpreter besitzt. Mit der interpretierten Programmiersprache Python lassen sich über recht einfache Funktionen schnell auch komplexere Aufgaben bearbeiten. In FreeCAD kann mit Python direkt in der GUI geskripted werden oder es lassen sich, so wie es im Folgenden beschrieben wird, externe Skripte erstellen, die sogar ohne eine geöffnete FreeCAD-Instanz den vollen Funktionsumfang nutzen können.

Eine sehr gute Einführung in das FreeCAD-Scripting bietet hierfür das FreeCAD-Wiki, dabei liefert für diese Arbeit insbesondere der Artikel »Topological data scripting« (2019) wichtige Informationen. Komplexere Funktionen können dagegen direkt aus der Python-Konsole in FreeCAD abgelesen werden. Jede Aktion, die der Benutzer über die GUI durchführt, wird als Python Code auf die Konsole ausgegeben.

Das G2FreeCAD Tool stellt daher genaugenommen kein eigenständiges Übersetzungstool dar, sondern ist nur ein Python-Skript, dass die Übersetzung steuert. Da aber wie schon in Abschnitt 3.3 erwähnt das Graphensystem GrGen.NET ausschließlich in C# gehandhabt wird, muss für dieses Tool eine eigene C#-Bibliothek (`GrGen2Py.dll`) entwickelt werden, welche die Informationen aus dem Graphen für das Python-Skripting verfügbar macht.

4.3.1 Grundstruktur des Programms

Das Python Programm wird in drei Dateien unterteilt, in das eigentliche Hauptskript `Translator.py` und die beiden Dateien `Entity.py` und `Constraint.py` in denen die entsprechenden Klassen zu den Knoten und Kanten des Graphensystems definiert werden. In dem Hauptskript soll dann zunächst der Graph eingelesen, dann die entsprechenden Klassen instanziert und schließlich mit den jeweiligen Methoden die FreeCAD-Geometrien gezeichnet werden, ganz analog zum zuvor vorgestellten Siemens NX Tool (vgl. Abschnitt 4.2.1).

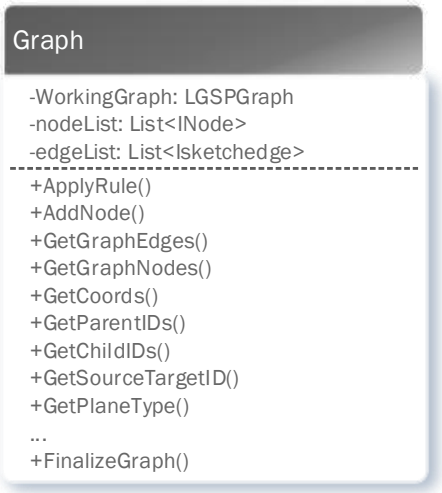


Abbildung 4.7: Graph-Klasse der C#-Bibliothek.

Da alle Funktionen der C#-Bibliothek in allen drei Dateien zur Verfügung stehen sollen und das Hauptskript auf alle Klassen Zugriff haben soll wird in `Translator.py` alles aus `Constraint.py` importiert und in `Constraint.py` alles aus `Entity.py`. In `Entity.py` wird dann das Modul **pythonnet** geladen (vgl. »Python for .NET«, 2019), womit eine Referenz zu der C#-Bibliothek erstellt werden kann, sowie die Python-Module `FreeCAD`, `Part` und `Sketcher`, die mit der Software FreeCAD mitgeliefert werden.

In `GrGen2Py.dll` wird nur eine Klasse `Graph` definiert (vgl. Abb. 4.7), die beim Instanzieren ähnlich der Hauptfunktion des Siemens NX Tools (vgl. Abschnitt 4.2.1) zunächst einen `StreamWriter` initialisiert. Auch hier wird damit eine Log-Datei erstellt, in der Fehler, Warnungen und abgeschlossene Aktionen festgehalten werden. Als nächstes wird das Graphensystem initialisiert, damit über das Python-Skript der Arbeitsgraph erstellt werden kann. Hier kann später die direkte Erstellung im Skript durch eine Importfunktion ersetzt werden, zur Entwicklung des Tools ist die direkte Erstellung aber vorteilhafter. Egal auf welche Weise der Graph erstellt bzw. eingelesen wurde, können mit den beiden Methoden `GetGraphNodes()` und `GetGraphEdges()` im Python-Skript dann die Namen der entsprechenden Knoten- und Kantentypen aus dem Arbeitsgraphen als *string* ausgelesen werden. Mit diesen Namen werden im Anschluss dann die entsprechenden Klassen instanziert.

In Abb. 4.8 ist die implementierte Klassenstruktur entsprechend der Knoten dargestellt, hierfür sind allerdings nur die wichtigsten Attribute und Methoden aufgelistet. Es wird eine Oberklasse `Entity` definiert, von der alle weiteren Klassen dieses Typs abgeleitet werden, die aber selbst nicht instanziert werden soll. Eine solche Klasse wird als abstrakt bezeichnet und kann in Python nur durch das Laden eines zusätzlichen Moduls erzeugt werden. Davon wird hier der Einfachheit halber aber abgesehen und später darauf geachtet, dass keine Entity-Instanz erstellt wird.

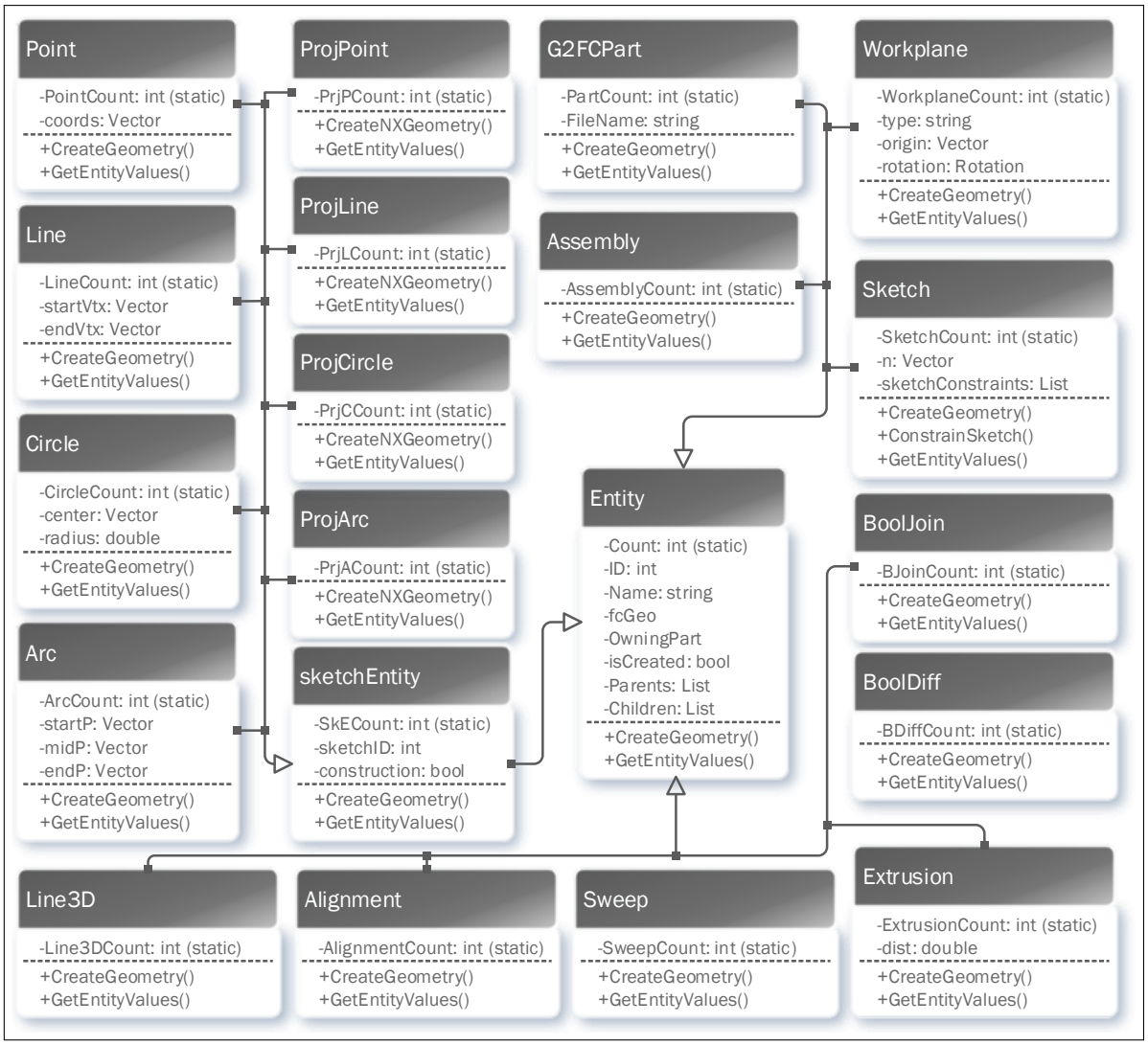


Abbildung 4.8: Klassenstruktur der Knoten. Von links anschließende Unterklassen entsprechen hierbei SketchNodes. Die von oben und unten anschließenden Unterklassen stellen die ProceduralNodes dar. Es sind nur die wichtigsten Attribute und Methoden abgebildet.

Die zwischengeschaltete Unterklasse `sketchEntity` wird eingeführt um zwei weitere Attribute zu definieren, die nur Linienobjekte betreffen, `construction` und `sketchID`. Mit dem ersten wird das das Umwandeln von Linienobjekten in Konstruktionslinien gesteuert und das zweite stellt später die ID dar, mit der in FreeCAD Sketch-Elemente angesprochen werden können.

Jede Entity-Klasse verfügt über eine eigene Erstellungsmethode `CreateGeometry()` zum Erstellen der FreeCAD Geometrien sowie über eine eigene `GetEntityValues()` Methode, mit der aus dem Graphensystem die für die entsprechende Entity-Klasse wichtigen Attribute extrahiert werden können. Zudem besitzt jede Klasse eine eigene statische Counter-Variable, mit der ganz analog zum Siemens NX Tool eindeutige Namen und IDs erzeugt werden.

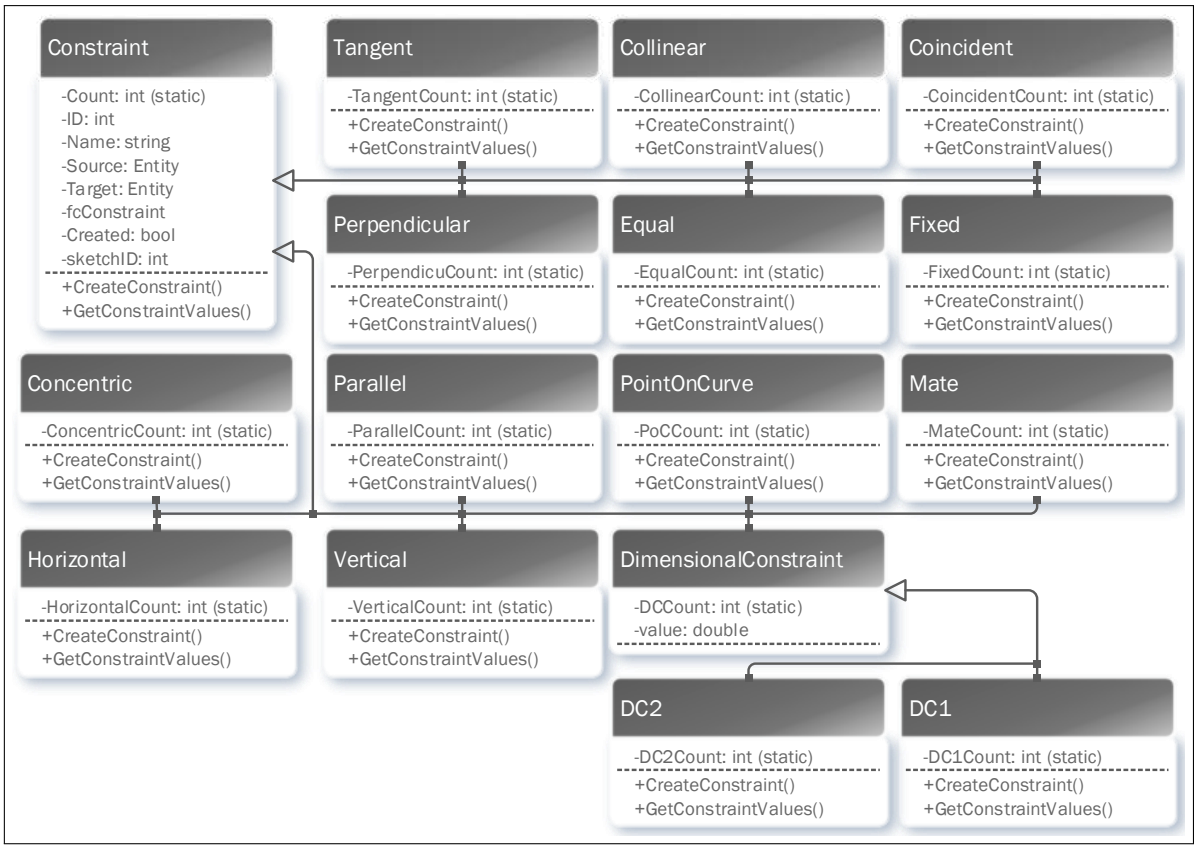


Abbildung 4.9: Klassenstruktur der Knoten. Auch hier sind nur die wichtigsten Attribute und Methoden abgebildet.

Die implementierte Klassenstruktur der Kanten ist in Abb. 4.9 dargestellt, auch hier wurden nur die wichtigsten Attribute und Methoden aufgelistet. Alle Unterklassen desselben Typs leiten sich hier von der Oberklasse `Constraint` ab, die ebenso wie die Klasse `Entity` als abstrakt angenommen aber nicht als abstrakt realisiert wird. Ganz analog gilt hier, dass jede Unterklasse eine eigene Erstellungsmethode sowie eine Methode zur Extraktion der Attributwerte definiert. Die zwischengeschaltete Unterklasse `DimensionalConstraint` wird definiert um später die beiden Constraints `dc1` und `dc2` gebündelt ansprechen zu können. Gleichzeitig kann hier auch das gemeinsame Attribut `value` eingeführt werden.

Es ist an dieser Stelle zudem zu erwähnen, dass nicht alle im Graphensystem definierten Zwangsbedingungen in FreeCAD direkt umsetzbar sind. In manchen Fällen muss daher in den entsprechenden Klassen eine Hilfskonstruktion gebildet werden für die weitere Attribute nötig sind, genaueres dazu wird in Abschnitt 4.3.3 erläutert.

4.3.2 Ablauf der Übersetzung

Genau wie schon in Abschnitt 4.2.2 für das Siemens NX Tool gezeigt, muss auch bei FreeCAD auf eine bestimmte Reihenfolge der Ausführung geachtet werden, dabei aber eine gewis-

se Adaptivität mit eingebaut werden. Im Gegensatz zum NX Tool wird hier keine extra `RelationBuilder`-Klasse implementiert, da die nötigen Codesegmente in Python sehr einfach und übersichtlich sind. An dieser Stelle wird daher der Reihe nach auf die einzelnen Statements im Hauptskript `Translator.py` eingegangen.

Nach dem Instanzieren der Graph-Klasse und dem Erstellen bzw. Einlesen des zu übersetzenden Graphen gliedert sich der implementierte Ablauf des Übersetzungstools wie folgt:

1. Auslesen der Knoten und Kanten.

```
nodes = graph.GetGraphNodes()
edges = graph.GetGraphEdges()
```

2. Erstellen der entsprechenden Python Objekte.

```
constraintList = []
entityList = []
for nStr in nodes:
    entityList.append(CreateEntityFromString(nStr))
for eStr in edges:
    constraintList.append(CreateConstraintFromString(eStr))
```

3. Abspeichern der relevanten Informationen in den Objektattributen.

```
for e in entityList:
    try:
        e.GetEntityValues(graph)
    except ProceduralError as err:
        print(err)
for c in constraintList:
    try:
        c.GetConstraintValues(graph)
    except ProceduralError as err:
        print(err)
```

4. Erzeugung der FreeCAD Skizzen samt sketchEntities und sketchConstraints.

```
sketches = [x for x in entityList if x == Sketch]
for sk in sketches:
    try:
        if not sk.isCreated:
            sk.CreateGeometry(graph)
    except ProceduralError as err:
        print(err)
```

5. Erzeugung aller nicht Skizzen-Geometrien und -Einschränkungen.

```
for e in entityList:
    try:
        if not e.isCreated:
            e.CreateGeometry(graph)
    except ProceduralError as err:
        print(err)
for c in constraintList:
    try:
        if not c.isCreated and c != Tangent:
            c.CreateConstraint(graph)
    except ProceduralError as err:
        print(err)
```

Unter Punkt 1 werden aus dem Graphen, der in dem GrGen2Py-Objekt `graph` gespeichert ist, die Knoten und Kanten als Liste von strings ausgelesen. Die Reihenfolge der Listen bestimmt später auch die IDs der entsprechenden Objekte, die mithilfe der strings erstellt werden.

Instanziert werden die einzelnen Objekte unter Punkt 2. Dies wird bewerkstelligt, indem die Funktion `CreateEntityFromString()` bzw. `CreateConstraintFromString()` aufgerufen wird, die innerhalb einer if-Verzweigung die strings prüft und dann den entsprechenden Constructor aufruft. Die erstellten Objekte werden dann zur weiteren Verwendung in die Listen `entityList` und `constraintList` gespeichert. Zusätzlich werden die Entity-Objekte jeweils in der statischen Variable `EntityList` der beiden Oberklassen `Entity` und `Constraint` gespeichert, um innerhalb der Methoden dieser Klassen die Entities leicht verfügbar zu machen.

Noch sind die erstellten Objekte aber nicht verwendbar, da die nötigen Informationen noch nicht an sie übertragen wurden. Dies geschieht unter Punkt 3 indem für jedes Objekt die eigene `GetEntityValues()` bzw. `GetConstraintValues()` Methode aufgerufen wird. Für alle Unterklassen gleich anzuwendende Statements werden in der Oberklasse definiert und in der Methode der Unterklassen als erstes aufgerufen. Anschließend werden alle weiteren nötigen Statements definiert. Dabei wird über die Graph-Methoden aus der C#-Bibliothek auf die Informationen im Graphen zugegriffen und die Werte entsprechend in den Objektattributen gespeichert. Eine Auflistung dieser Methoden und deren Rückgabewerte ist in Tabelle 4.1 gegeben.

Nachdem alle nötigen Informationen eingeholt wurden kann mit der Erstellung der Geometrien begonnen werden. Hierzu werden unter Punkt 4 ganz analog zum Siemens NX Tool zunächst alle Skizzen erstellt (vgl. Abschnitt 4.2.2 – CreateSketchGeometry()). In allen `CreateGeometry()`-Methoden wird zunächst geprüft ob die Parent-Geometrien existieren und ggf. zuvor erstellt, so dass alle Abhängigkeiten eingehalten werden. Mit der Skizzen-Erstellungsmethode werden

Tabelle 4.1: Graph-Methoden zum Auslesen der Knoten- und Kanteninformationen.

Methode	Rückgabewert
`GetParentIDs()`	IDs der Parents mithilfe einer Extension-Methode
`GetChildIDs()`	IDs der Childs mithilfe einer Extension-Methode
`GetCoords()`	Alle nötigen Koordinaten mithilfe einer Überladenen Extension-Methode
`GetPlaneType()`	WorkPlane Orientierung als string
`GetConstruction()`	Boolscher Wert, gibt an ob das Objekt eine Konstruktionslinie darstellt
`GetExtrusionDist()`	Extrusionslänge als double
`GetSourceTarget()`	Ursprung- und Zielknoten IDs einer Kante
`GetValue()`	Wert eines dimensional Constraints als double

zudem gleich nach der eigenen Erzeugung auch alle Skizzen-Elemente und anschließend die entsprechenden Zwangsbedingungen erstellt.

Unter Punkt 5 wird die Übersetzung dann abgeschlossen, indem alle restlichen Geometrien und Zwangsbedingungen erstellt werden. Falls zuvor keine Fehler passiert sind, sollten an dieser Stelle nur noch die 3D-Operationen bei den Geometrien und Baugruppeneinschränkungen bei den Zwangsbedingungen fehlen. Bei FreeCAD gibt es allerdings einige Probleme bei einigen dieser Objekte, insbesondere im Zusammenhang mit Baugruppen, da dafür kein offizielles Toolkit zur Verfügung steht. Mehr Informationen hierzu werden im folgenden Abschnitt (4.3.3) gegeben.

4.3.3 Probleme bei der Implementierung

Obwohl der Umgang mit der API von FreeCAD sehr intuitiv und komfortabel ist, mangelt es an bestimmten Stellen an Funktionsumfang. Manche dieser Mängel sind leicht durch etwas extra Programmieraufwand umgehbar, andere jedoch sind so gravierend, dass das Übersetzungstool nicht zu 100% umgesetzt werden kann. Im Folgenden werden diese Probleme mit aufsteigendem Schweregrad aufgelistet.

Fehlende Zwangsbedingungen

Die FreeCAD Sketcher Toolbox ist aufgrund des Open Source Charakters dieser Software nicht besonders umfangreich. Ein paar der Einschränkungen, die in dem verwendeten Metamodel definiert werden, existieren daher nicht in FreeCAD, oder sind nicht entsprechend anwendbar. Diese Probleme können allerdings alle mit ein wenig Aufwand umgangen werden.

Folgende Einschränkungen aus dem Graphensystem sind hier demnach problematisch:

- Concentric
- Collinear
- Horizontal/Vertical
- dc2

Die ersten beiden genannten Einschränkungen lassen sich hier aber sehr einfach substituieren. Anstatt zwei Kreise zueinander konzentrisch zu setzen, können auch die Mittelpunkte koinzident gesetzt werden. Genauso können Linien, die kollinear sein sollen, als zueinander tangent festgelegt werden. Für die letzten beiden genannten Einschränkungen müssen allerdings Hilfsgeometrien erzeugt werden, um den gewünschten Effekt zu erzielen.

Bei Horizontal bzw. Vertikal ist an dieser Stelle nur die räumliche Anordnung (engl. *Alignment*) gemeint, wenn im Graphensystem zur Kante inzidente Knoten unterschiedlich sind, also die Zwangsbedingung auf zwei Objekte angewendet wird. Um dies zu bewerkstelligen wird einfach eine Hilfskonstruktionslinie zwischen den beiden Objekten gezeichnet und mit der entsprechenden Einschränkung (*horizontal* oder *vertical*) belegt.

Bei der dimensionalen Einschränkung ist dieses Problem etwas vielschichtiger, da hier abhängig von den Geometrien, die eingeschränkt werden sollen, unterschiedliche Szenarien auftreten können. Die einzigen Szenarien, die ohne Hilfsgeometrien abgebildet werden können, sind der Abstand zweier Punkte oder der Abstand eines Punktes zu einem anderen Linienobjekt. Der Abstand zweier Linienobjekte zueinander ist etwas umständlicher zu gestalten, hierbei ist es auch wichtig zu unterscheiden welche Arten von Linienobjekten zueinander bemaßt werden (z.B. Abstand zweier Linien, Abstand zweier Kreise oder eine Kombination aus beidem). In jedem Fall wird aber eine Hilfskonstruktionslinie erstellt, deren Länge bemaßt wird und die entsprechend der Situation zusätzlich mit Hilfsconstraints belegt wird.

Fehlender Geometrie Port

Im Graphensystem wird davon ausgegangen, dass eine Linie einen Mittelpunkt besitzt, jedoch existiert dieser Port in FreeCAD nicht und muss durch Hilfsgeometrien künstlich erzeugt werden. Hierzu werden zwei extra Hilfskonstruktionslinien gezeichnet, die zueinander gleich lang, kollinear zur Hauptlinie und über eine entsprechende Koinzidenz der Randpunkte genau in die Hauptlinie eingepasst sind (vgl. Abb. 4.10). Der auf diese Weise erzeugte Mittelpunkt kann dann wie gewohnt für alle betroffenen Einschränkungen verwendet werden.

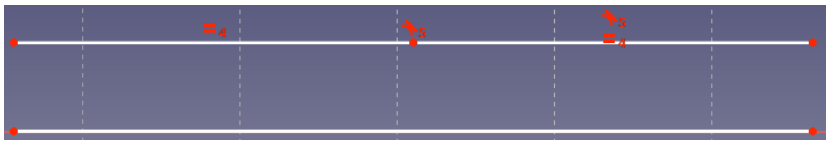


Abbildung 4.10: Über Hilfsgeometrien und -einschränkungen erzeugter Linienmittelpunkt. Zum Vergleich Linie unten ohne Hilfselemente.

Fehlende Intersection Tools

Das Objekt `ProjectedPoint` basiert darauf, dass der Schnittpunkt einer im Raum liegenden Kurve mit der Sketchebene gefunden wird. In FreeCAD existiert aber bedauerlicherweise kein Tool, welches diese Operation durchführen kann. Daher muss ein Umweg gefunden werden, der ein ähnliches Ergebnis erzielt. Im Tunnelbauwerk-Beispiel dieser Arbeit wird das Alignment, die Kurve entlang welcher der Tunnel gezeichnet werden soll, in die Skizze projiziert und an dem daraus resultierenden Punkt die gesamte Skizze ausgerichtet. Der Einfachheit halber ist das Alignment in dieser Arbeit eine gerade Linie.

Ganz allgemein kann ein Schnittpunkt einer Gerade mit einer Ebene mithilfe der parametrischen Geradengleichung und der Normalenform der Ebenengleichung berechnet werden. Die Berechnung wird wie in Abb. 4.11 veranschaulicht durchgeführt. Sieht man die Abbildung im Kontext der ProjectedPoint Problemstellung, so stellt die Ebene E die Skizze dar mit seinem eigenen lokalen Koordinatensystem und liegt zusammen mit dem Alignment (Geradenstück) im Raum des übergeordneten Parts mit dessen globalen Koordinatensystem.

Algorithmus 4.8 zeigt die implementierte `CreateGeometry()`-Methode für die ProjPoint-Klasse, die zuvor beschriebene Berechnung findet hier in den Zeilen 11–14 statt. Diese Methode funktioniert momentan nur mit einer Gerade als Alignment, lässt sich aber später auch für andere Kurven realisieren, sofern eine Parametergleichung verfügbar ist. Außerdem

I) Ebene: $\mathbf{n} \cdot (\mathbf{x} - \mathbf{a_E}) = \mathbf{0}$
II) Gerade: $\mathbf{a_g} + \lambda \cdot \mathbf{b} = \mathbf{x}$

II) in I) $\lambda_{SP} = \dfrac{-(\mathbf{a_g} - \mathbf{a_E}) \cdot \mathbf{n}}{\mathbf{b} \cdot \mathbf{n}}$

$\Rightarrow$ $\mathbf{x_{SP}} = \mathbf{a_g} + \lambda_{SP} \cdot \mathbf{b}$

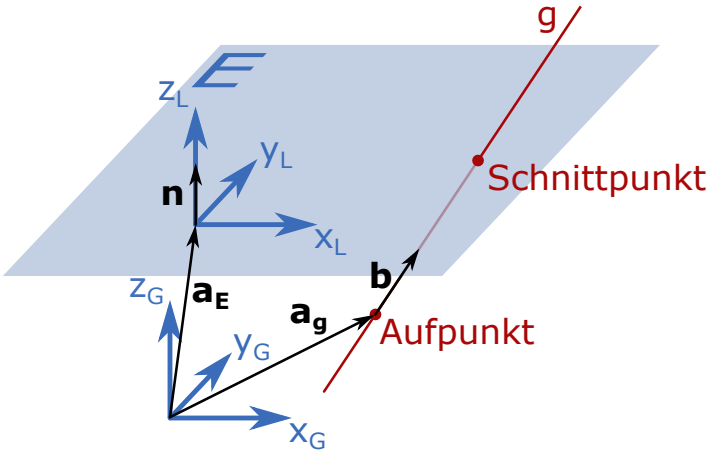


Abbildung 4.11: Bestimmung des Schnittpunktes von einer Geraden mit einer Ebene.

Algorithmus 4.8: CreateGeometry-Methode der ProjPoint-Klasse

```
def CreateGeometry(self, graph):
    try:
        super(ProjPoint, self).CreateGeometry(graph)
        if self.isCreated:
            return

        sk = self.OwningSketch
        pE = next((p for p in self.Parents if type(p) is Alignment),
                  None)

        aE = sk.fcGeo.Placement.Base
        w = pE.startVtx - aE  # ag - aE
        l = -w.dot(sk.n) / pE.u.dot(sk.n)
        xSP = w + l*pE.u

        self.fcGeo = Part.Point(xSP)
        self.sketchID = sk.fcGeo.addGeometry(self.fcGeo)
        self.skName += str(self.sketchID + 1)

        sk.fcGeo.addConstraint(Sketcher.Constraint('DistanceX',
            self.sketchID, 1, xSP.x))
        sk.fcGeo.addConstraint(Sketcher.Constraint('DistanceY',
            self.sketchID, 1, xSP.y))
        self.FinalizeGeometry(graph)
    except Exception as ex:
        print(ex)
        self.SendError(graph)
```

wird nicht explizit getestet ob der Schnitt überhaupt möglich ist (Alignment parallel zur Ebene), es würde dann eine Nulldivision auftreten, die aber von der try-except-Umgebung abgefangen wird.

Um sicherzustellen, dass der Punkt an der berechneten Stelle bleibt wird nach der Erstellung noch fixiert mit den Constraints *DistanceX* und *DistanceY*, entsprechend der Graph-Kante `equal`. Ein Nachteil an diesem Algorithmus ist, dass bei einer Lageänderung des Alignments der Punkt nicht automatisch mitbewegt wird. Eine solche Änderung sollte aber ohnehin nicht auftreten bzw. wenn doch, dann muss die gesamte Geometrie so oder so neu gezeichnet werden.

Fehlerhaftes Sweeping Tool

Das im Standard Toolset Part enthaltene Sweeping-Tool funktioniert bedauerlicherweise nicht in allen Fällen korrekt. Es werden genau dann falsche Geometrien erzeugt, wenn mehr als eine geschlossene Kurve im zu sweependen Sketch enthalten ist, z.B. bei Rohrgeometrien. Genau das ist aber der Fall bei dem Tunnelbauwerk-Beispiel. Die erstellten Sweeps sind in diesen Fällen nicht hohl, d.h. die innenliegende geschlossene Kurve wird nicht berücksichtigt.

Um dieses Problem mit diesem Sweep-Tool zu umgehen müsste das Objekt komplett anders im Graphensystem modelliert werden, im Falle eines Rohres müssten zwei Vollrohre erstellt werden mit unterschiedlichen Radien und voneinander mit der *BoolDiff*-Operation abgezogen werden. Alternativ gibt es auch noch eine weitere mitgelieferte Toolbox, den **PartDesigner**, hier ist die Dokumentation aber leider sehr lückenhaft – auch die Ausgabe der Python-Konsole ist nicht ausreichend und zudem müsste ein großer Teil der Programm-Grundstruktur überarbeitet werden. Es wird hier also davon abgesehen diese Methode weiter anzupassen, dies ist aber grundsätzlich möglich.

Kein offizielles Tool zur Assembly Erstellung

Dies ist das größte Problem bei der Entwicklung des Übersetzungstools, denn deswegen kann momentan unter keinen Umständen die Übersetzung von Graphen, die Baugruppen darstellen, ermöglicht werden. Es gibt zwar Toolsets, mit denen Baugruppen erstellt werden können, jedoch sind diese von der FreeCAD-Community entwickelt worden und daher zum einen nicht 100% verlässlich und schlimmer noch, überhaupt nicht dokumentiert, d.h. auch keine Ausgabe auf die Python-Konsole.

Kapitel 5

Ergebnisse

Um die Funktionsweise der implementierten Programme zu beurteilen werden im Folgenden die beiden in der Arbeit genannten Beispiele mit beiden Übersetzungstools jeweils ausgeführt und die resultierenden Modelle evaluiert.

Das Siemens NX Tool wird wie in Abschnitt 3.2 gestartet indem in einer offenen NX Instanz über die Menüoption „`File -> Execute -> NX Open...`“ das Programm ausgewählt wird. Nach dem Starten erstellt das Programm eigenständig sequenziell die im Graphen definierten Objekte. Während der Ausführung werden alle neu gezeichneten Geometrien sofort gerendert, so dass der Erstellungsprozess direkt mitverfolgt werden kann. Nach Beenden des Programms bleibt die NX Instanz weiter geöffnet und das erstellte Modell kann direkt begutachtet werden. Die NX-Bauteil-Dateien werden direkt im Verzeichnis des Übersetzungstools gespeichert.

Das FreeCAD Tool dagegen ist wie in Abschnitt 3.3 beschrieben ein Python Skript, dass über eine geeignete Python-IDE ausgeführt werden kann. Startet man das Skript so werden alle Operationen im Hintergrund durchgeführt, d.h. der Erstellungsprozess ist für den Anwender nicht sichtbar. Als Ergebnis liefert das Skript entsprechend FreeCAD-Bauteil-Dateien im Verzeichnis des Python Skripts, die nach Beenden des Skriptes über die FreeCAD-Software begutachtet werden können.

5.1 IPE-Träger

In diesem Beispiel soll ein einfacher IPE-Träger gezeichnet werden. Hierzu soll eine Baugruppe erstellt werden, die ein Bauteil beinhaltet, welches wiederum eine Skizze beinhaltet. Aus dieser Skizze wird dann über die 3D-Operation Extrusion ein 3D-Körper erstellt. Beide Übersetzungstools liefern für dieses Beispiel genau das richtige Ergebnis.

Mit beiden Übersetzungstools wurden auch jeweils Log-Dateien ausgegeben, mit denen der jeweilige Erstellungsprozess nachvollzogen werden kann. Diese Log-Dateien sowie die entsprechenden Bauteil-Dateien für beide Systeme sind im Dateianhang dieser Arbeit zu finden.

G2NX

In Abb. 5.1 ist der vom NX Tool erstellte IPE-Träger zu sehen. Zur besseren Veranschaulichung ist der Träger so gedreht, dass die Unterseite zu sehen ist, zudem ist die Skizze überlagert eingeblendet. In der Skizze sind aufgrund des Zoomgrades die Zwangsbeziehungen teilweise ausgeblendet. In der Statusleiste des NX-Fensters ist aber zu sehen, dass die Skizze mit nur zwei *auto dimensions* vollständig eingeschränkt ist.

Die zwei *auto dimensions* sind in diesem Fall zwei weitere dimensionale Bemaßungen, die theoretisch nötig sind, dass die Elemente komplett unverschieblich sind. Relativ zueinander sind die Elemente zwar fixiert, die Gesamtskizze ist aber relativ zum Koordinatensystem weiterhin verschieblich. Dies ist aber nur dann ein Problem, wenn dieses Bauteil in einer Baugruppe verwendet wird und nachträglich die Größe verändert wird und zudem kein Fehler des Übersetzungsprogramms, da diese Bemaßungen nicht in der Graphendarstellung definiert wurden.

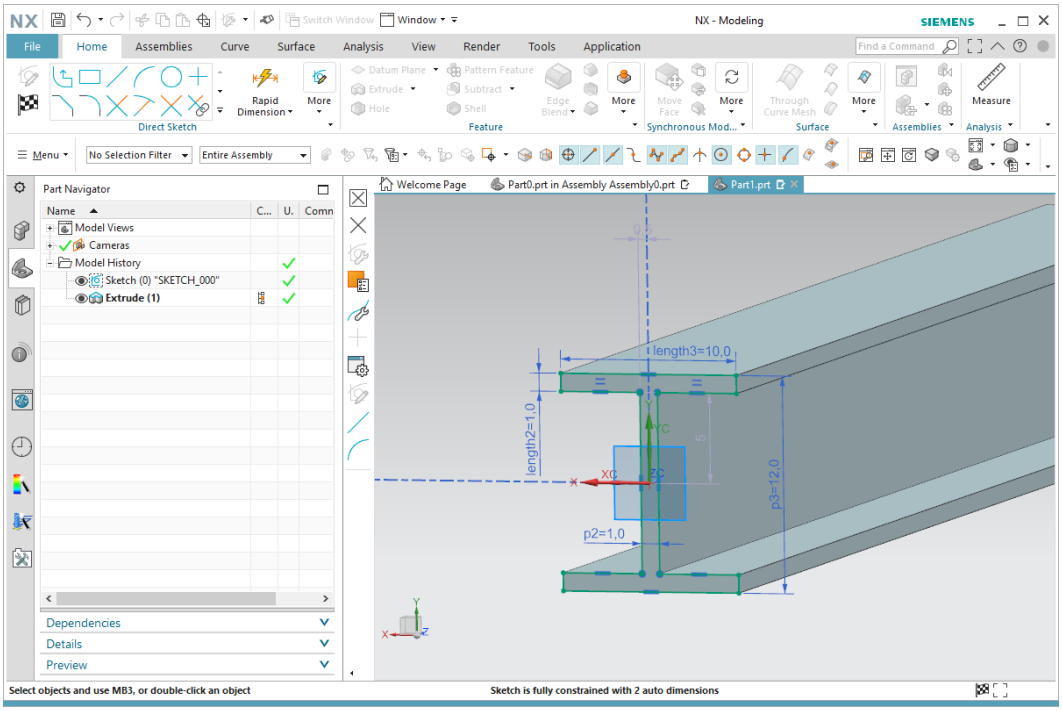


Abbildung 5.1: IPE-Träger erstellt mit G2NX

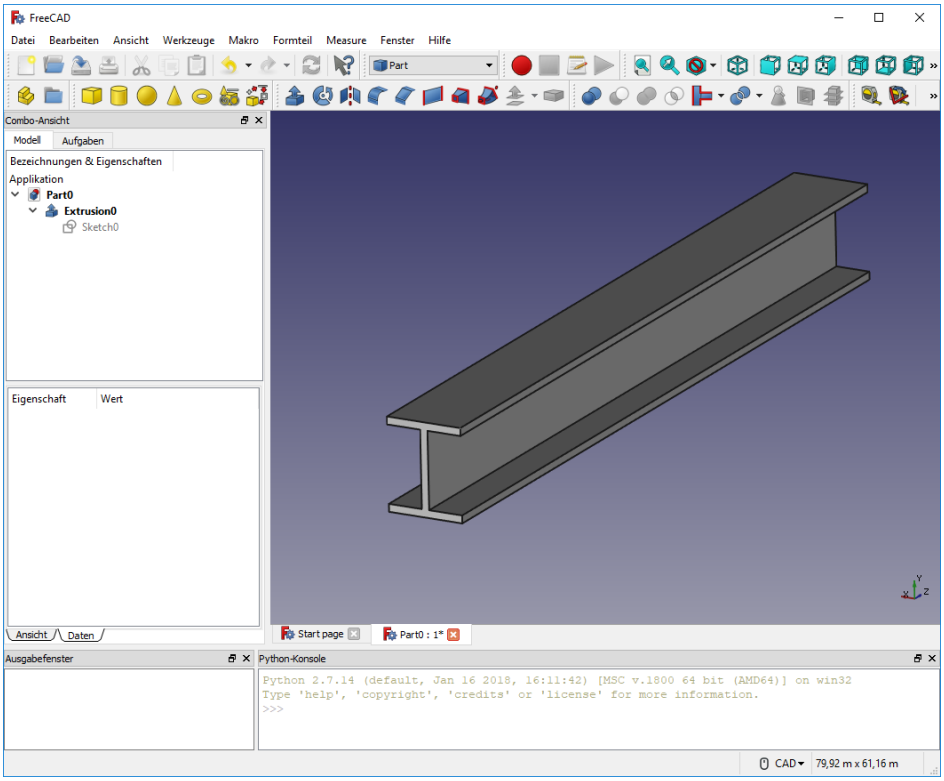


Abbildung 5.2: IPE-Träger erstellt mit G2FreeCAD

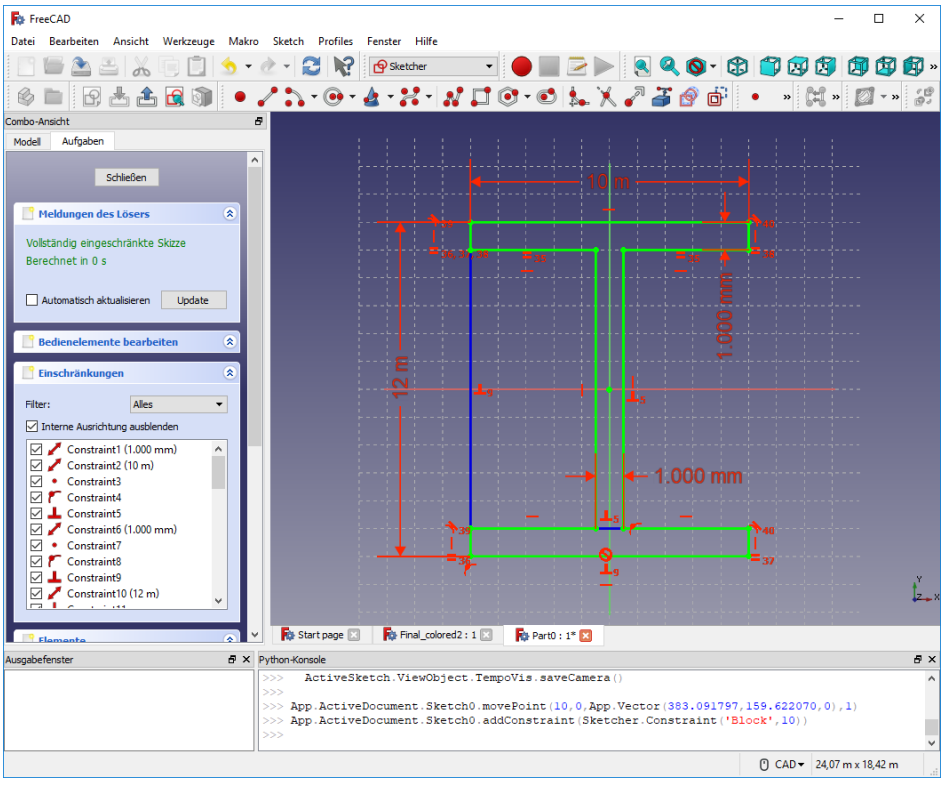


Abbildung 5.3: Skizze für den IPE-Träger

G2FreeCAD

In Abb. 5.2 und 5.3 ist der durch das FreeCAD Tool erstellte IPE-Träger und dessen Skizze dargestellt. Es ist in Abb. 5.2 zudem zu erkennen, dass das Objekt zu einer Bauteil-Datei gehört, ein Sketch vorhanden ist und mit diesem der 3D-Körper `Extruion0` erstellt wurde. Eine Baugruppe existiert hier aber nicht, da wie schon in Abschnitt 4.3.3 beschrieben kein offizielles Baugruppen-Tool für FreeCAD existiert.

Abbildung 5.3 zeigt hier eine vollständig Eingeschränkte Skizze, jedoch wurde hier etwas nachgeholfen und die unterste Linie nachträglich blockiert (symbolisiert durch den durchgestrichenen Kreis). Der Python-Code für diese Blockieren Einschränkung ist zudem in der Python-Konsole auf dem Bild sichtbar.

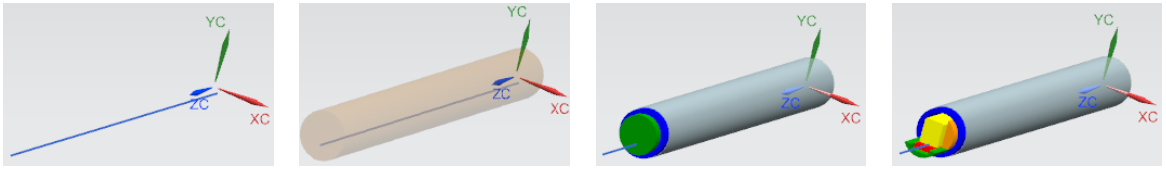
Die blauen Linien stellen hier Konstruktionslinien dar und sind die in Abschnitt 4.3.3 vorgestellten Hilfsstrukturen für die `dc2`-Bedingung. Links im Bild wird auch eine Liste aller Einschränkungen abgebildet, hier kann man auch erkennen in welcher Reihenfolge die entsprechenden Zwangsbedingungen erstellt wurden.

5.2 Tunnelbauwerk

In diesem Beispiel soll ein Tunnelabschnitt in Detailstufe LoD4 modelliert werden. Hierzu soll eine Baugruppe erstellt werden, die ein Bauteil enthält, das wiederum acht Skizzen enthält, die alle entlang desselben *Alignments* „gesweeped“ werden. Zudem sind die Skizzen diesem Beispiel voneinander abhängig, da bestimmte Elemente projiziert werden. Anzumerken ist, dass anhand des Graphen in Detailstufe LoD4 auch alle niedrigeren Detailstufen enthalten sind (vgl. Abb. 5.4). Auch für dieses Beispiel wurden jeweils Log-Dateien erstellt, mit denen der Erstellungsprozess nachvollzogen werden kann.

G2NX

In Abb. 5.5 ist die Lösung des Siemens NX Tools dargestellt. Zur besseren Visualisierung sind die einzelnen Komponenten nachträglich noch zugeschnitten und eingefärbt worden. Alle 3D-



LoD1 LoD2 LoD3 LoD4

Abbildung 5.4: Detailstufen des Tunnelbauwerks

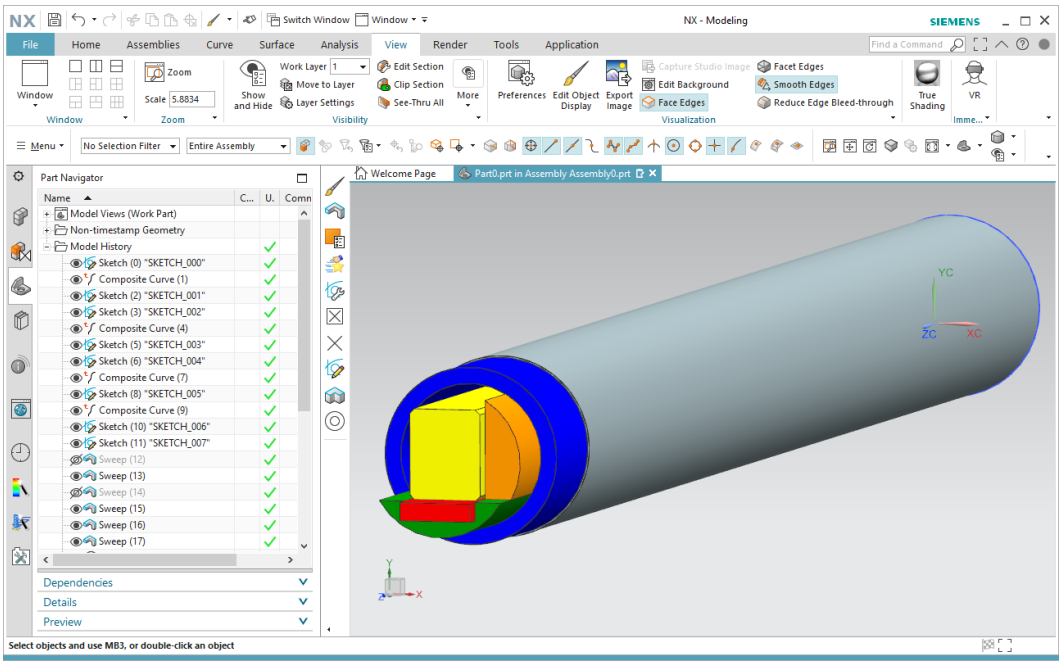


Abbildung 5.5: Tunnelbauwerk erstellt mit G2NX

Körper, die nicht zur Detailstufe LoD4 gehören, wurden zudem ausgeblendet. Alle Geometrien sind hier korrekt erstellt worden und mussten nicht nachträglich bearbeitet werden.

Selbst der *Service Space*, im Bild orange dargestellt, wurde problemlos erstellt. Wie schon in Abschnitt 2.1.4 erläutert hat die hier zugrundeliegende Skizze einige Probleme bereitet. Die einzige Möglichkeit diese Geometrie richtig zu erzeugen war letztendlich eine Modifikation am Graphensystem, so dass der Kreisbogen dieser Skizze besser platziert wird.

Auf der linken Seite in Abb. 5.5 werden unter dem Menüpunkt *Model History* alle gezeichneten Bauteil-Elemente zeitlich sortiert gelistet. Wie zu erkennen ist wurden zunächst alle Skizzen gezeichnet und erst im Anschluss nach und nach „gesweeped“. Zudem ist sichtbar, dass nach manchen Skizzen eine *Composite Curve* erstellt wurde. Diese Kurve stellt ein extrahiertes Feature aus der vorhergehenden Skizze dar und wird in eine andere nachfolgende Skizze projiziert.

G2FreeCAD

In Abb. 5.6 ist die Lösung des FreeCAD Tools dargestellt. Wie schon in Abschnitt 4.3.3 erläutert hat das Sweeping Tool von FreeCAD Probleme mit Skizzen, die mehrere geschlossene Linienführungen haben. Der im Bild sichtbare *Lining Space*, in blau dargestellt, ist per Hand nachbearbeitet worden aus einem Verschnitt von zwei Vollrohren, dem *Full Tunnel Space* (orange dargestellt in LoD2 in Abb. 5.4) und dem *Interior Space* (grün dargestellt in LoD3 in

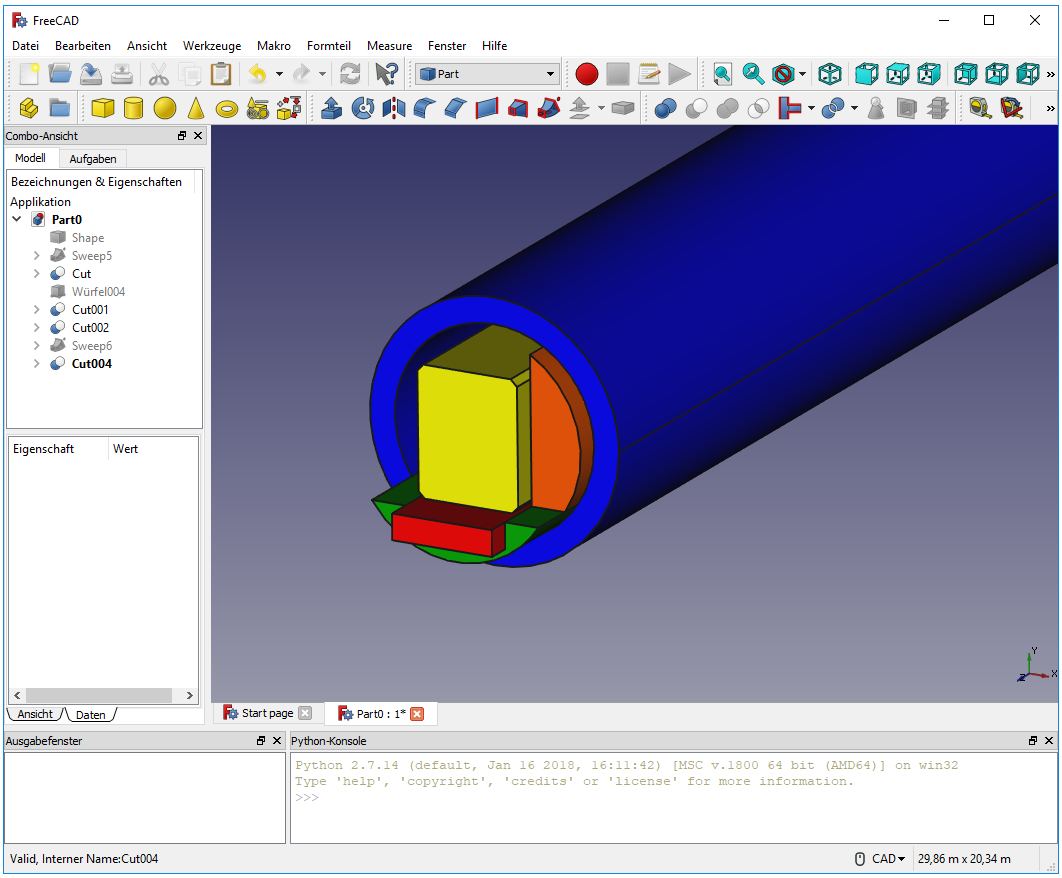


Abbildung 5.6: Tunnelbauwerk erstellt mit G2FreeCAD

Abb. 5.4). Ansonsten wurden auch hier alle weiteren für die Detailstufe LoD4 nicht relevanten Geometrien ausgeblendet.

Gut zu erkennen ist jedoch, dass auch mit dem FreeCAD Tool der *Service Space*, im Bild orange dargestellt, korrekt gezeichnet wurde. Auch die anderen Skizzen weisen keinerlei Fehler auf und sind alle vollständig eingeschränkt, so wie es sein soll. Auf der linken Seite im Bild werden auch wieder alle erstellten Bauteil-Elemente aufgelistet, allerdings wird hier keine zeitliche Reihenfolge dargestellt, sondern nach direkten Abhängigkeiten sortiert, d.h. unter 3D-Operationen sind die damit bearbeiteten Körper untergeordnet und unter den Körpern sind die Skizzen untergeordnet, aus denen sie erstellt wurden.

Kapitel 6

Zusammenfassung und Ausblick

Das fertige Produkt dieser Arbeit stellen zwei funktionsfähige Übersetzungsprogramme dar, die aus einer graphenbasierten Darstellung eines Modells ein parametrisches Modell erzeugen können. Zum einen das Tool *G2NX* für die CAD-Software Siemens NX und zum anderen das Tool *G2FreeCAD* für dessen Open Source Äquivalent FreeCAD.

In Anbetracht der Ergebnisse kann gesagt werden, dass sich mit beiden Übersetzungstools graphenbasierte Darstellungen von Modellen erfolgreich interpretieren lassen. Beide Tools liefern in beiden dargestellten Beispielen sehr zufriedenstellende Lösungen. Um den Funktionsumfang weiter zu testen müssten aber noch weitere Graphenbeispiele angefertigt werden, was den Umfang dieser Arbeit aber deutlich überschritten hätte. Des weiteren reicht der bisher implementierte Funktionsumfang der Übersetzungsprogramme noch nicht aus, um praxisnahe Beispiele zu testen.

Allerdings ist das FreeCAD Tool nicht vollkommen zuverlässig und bedarf noch weiterer Überarbeitung, vor allem im Hinblick auf die Erstellung der 3D-Körper und die Erzeugung von Baugruppen. Dies ist aber darin zu begründen, dass FreeCAD ein Open Source Software-Projekt ist und in der Entwicklung das nötige Fachpersonal fehlt. Bestimmter Funktionsumfang fehlt hier entweder ganz oder wurde nicht komplett fehlerfrei implementiert. Die Erzeugung von Baugruppen z.B. wird erst realisierbar sein, sobald ein zuverlässiges offizielles Baugruppentool für FreeCAD veröffentlicht wird.

Bei der Entwicklung des Siemens NX Tools sind im Laufe der Arbeit keine großen Schwierigkeiten aufgetreten. Durch den großen Funktionsumfang der API konnten alle geforderten Mechanismen meist einfach und immer zuverlässig umgesetzt werden. Da die Generierung der parametrischen Darstellung aber in einer geöffneten NX-Instanz erfolgt und jedes Element sofort gerendert wird, dauert der komplette Übersetzungsprozess unnötig lange. Selbst in den beiden hier dargestellten einfachen Beispielen, bei denen vergleichsweise wenige Elemente involviert sind, dauert der Übersetzungsvorgang mehrere Sekunden. Bei deutlich komplexe-

ren Modellen ist daher davon auszugehen, dass sich eine entsprechende Übersetzung sehr zeitintensiv gestaltet.

Für beide Tools kann allerdings gesagt werden, dass eine Übersetzung **nur dann** korrekt durchgeführt werden kann, wenn genügend viele Informationen im Graphen explizit mitgeliefert werden. Je weniger Informationen, wie z.B. die vorläufige Platzierung der einzelnen Geometrien, mitgegeben werden umso schwieriger wird es den Graphen zu interpretieren. Es ist also eine große Sorgfalt bei der Erstellung der graphenbasierten Darstellung und entsprechender Ersetzungsregeln gefordert. Eine umgekehrte Implementierung des Übersetzungsprogramms, d.h. die Übersetzung von parametrischer Darstellung in eine graphenbasierte Darstellung, könnte sich dabei als eine große Unterstützung erweisen.

Es kann also im Anbetracht der Ergebnisse bestätigt werden, dass eine Interpretation der in dieser Arbeit verwendeten Graphen für alle ähnlichen CAD-Systeme erfolgen kann. Somit kann z.B. eine solche Graphendarstellung auch als Austauschformat zwischen verschiedenen CAD-Softwarelösungen fungieren.

Anhang A

Datenpaket-Inhalt

In dem dieser Arbeit beigefügten Datenpaket ist folgender Inhalt gespeichert:

- Die vorliegende Bachelorarbeit im PDF-Format
- Der Quellcode der beiden Übersetzungstools
- Mit den Übersetzungstools erzeugte CAD-Dateien

Literatur

Blomer, J., Geiß, R. & Jakumeit, E. (2013). The GrGen . NET User Manual, 278.

Borrmann, A., Kolbe, T., Donaubauer, A., Steuer, H., Jubierre, J. & Flurl, M. (2014). Multi-Scale Geometric-Semantic Modeling of Shield Tunnels for GIS and BIM Applications. *Computer-Aided Civil and Infrastructure Engineering*, (30), 263–281.

Grzegorz, R. (1999). *Handbook Of Graph Grammars And Computing By Graph Transformations, Vol 2: Applications, Languages And Tools*. world Scientific.

Hack, S. (2003). Graphersetzung für Optimierungen in der Codeerzeugung. *Master's thesis, Universität Karlsruhe*.

Heckel, R. (2006). Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, *148*(1 SPEC. ISS.), 187–198. doi:10.1016/j.entcs.2005.12.018

NX. Answers for Industry. (2013). Zugriff 18. Oktober 2019 unter https://www.computerkomplett.de/assets/uploads/files/Produktbroschuere_NX_4639_tcm73-1423.pdf

Python for .NET. (2019). Zugriff 18. Oktober 2019 unter http://pythonnet.github.io/

Siemens. (2016). Getting Started with NX Open. (September). Zugriff unter https://docs.plm.automation.siemens.com/data_services/resources/nx/11/nx_api/common/en_US/graphics/fileLibrary/nx/nxopen/nxopen_getting_started_v11.pdf

Siemens Closes Acquisition of UGS. (2007). Zugriff 18. Oktober 2019 unter https://www.plm.automation.siemens.com/global/en/our-story/newsroom/siemens-press-release/43058

Siemens Documentation: Programming Tools. (2014). Zugriff 18. Oktober 2019 unter https://docs.plm.automation.siemens.com/tdoc/nx/10/nx_api#uid:index

Tittmann, P. (2019). *Graphentheorie. Eine anwendungsorientierte Einführung* (3., aktualisierte Auflage). Carl Hanser Verlag GmbH & Co. KG. doi:10.3139/9783446465039

Topological data scripting. (2019). Zugriff 18. Oktober 2019 unter https://www.freecadweb.org/wiki/Topological_data_scripting/de

Über FreeCAD. (2019). Zugriff 18. Oktober 2019 unter https://www.freecadweb.org/wiki/About_FreeCAD

Vajna, P. D. S., Weber, P. D. C., Zeman, P. D. K., Hehenberger, D.-I. P., Gerhard, U.-P. D. D. & Wartzack, P. S. (2018). *CAx für Ingenieure* (3. Auflage). Springer Berlin. doi:10.1007/978-3-662-54624-6

Vilgertshofer, S. (2014). *Repräsentation und Detaillierung parametrischer Skizzen mit Hilfe von Graphersetzungssystemen* (Magisterarb., Technische Universität München).

Vilgertshofer, S. & Borrmann, A. (2015). Automatic detailing of parametric sketches by graph transformation. *32nd International Symposium on Automation and Robotics in Construction and Mining: Connected to the Future, Proceedings*.

Vilgertshofer, S. & Borrmann, A. (2016). A graph transformation based method for the semi-automatic generation of parametric models of shield tunnels. *23rd International Workshop of the European Group for Intelligent Computing in Engineering, EG-ICE 2016*, 1–10.

Vilgertshofer, S. & Borrmann, A. (2017). Using graph rewriting methods for the semi-automatic generation of parametric infrastructure models. *Advanced Engineering Informatics*, *33*, 502–515. doi:10.1016/j.aei.2017.07.003

Vilgertshofer, S. & Borrmann, A. (2018). Supporting feature-based parametric modeling by graph rewriting. *ISARC 2018 - 35th International Symposium on Automation and Robotics in Construction and International AEC/FM Hackathon: The Future of Building Things*, (Isarc). doi:10.22260/isarc2018/0093

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelor-Thesis selbstständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ich versichere außerdem, dass die vorliegende Arbeit noch nicht einem anderen Prüfungsverfahren zugrunde gelegen hat.

München, 11. November 2019

Martin Slepicka

Martin Slepicka
[redacted]