Lehrstuhl für Computergestützte Modellierung und Simulation
Ingenieurfakultät für Bau Geo Umwelt
Technische Universität München

Eine objektorientierte Sprache zur Einbettung von Interpretationssemantik in digitale Bauwerksmodelle

Julian Amann

Vollständiger Abdruck der von der Ingenieurfakultät Bau Geo Umwelt der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. rer. nat. Ernst Rank
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. André Borrmann
	2.	Prof. Dr. rer. nat. Rüdiger Westermann
	3.	Prof. Dr. Jakob Beetz

Die Dissertation wurde am 25.09.2018 bei der Technischen Universität München eingereicht und durch die Ingenieurfakultät Bau Geo Umwelt am 10.12.2018 angenommen.

Zusammenfassung

Im Rahmen dieser Arbeit wird die zentrale Fragestellung untersucht, wie die wachsende Komplexität von Bauwerksinformationsmodellen beherrschbar gehalten werden kann. Dabei werden höherwertige semantische Konzepte für den Datenaustausch im Bereich des Building Information Modeling und deren Anwendungen für den Infrastrukturbau untersucht. Als Kernidee wird vorgeschlagen, Informationen über Bauwerke in Form von objektorientierten Programmen auszutauschen, um so Datenmodelle bezüglich ihrer Anforderungen flexibler zu gestalten und damit der steigenden Komplexität entgegenzuwirken. Hierzu wird eine dedizierte Programmiersprache (IFC-PL) zur Beschreibung von Programmen entwickelt, die in das bestehende Datenmodell Industry Foundation Classes (IFC) eingebettet werden kann. Anhand von konkreten Fallbeispielen wird die Anwendbarkeit des vorgestellten Ansatzes dargelegt. Dabei wird gezeigt, wie abstrakte Konzepte beschrieben werden können, die über die rein traditionelle Datenhaltung und -repräsentation von Bauwerksdaten hinausgehen, wie etwa die Prüfung von Normen und Richtlinien. Zudem werden alternative Ansätze und Konzepte aufgezeigt, die eine flexible Erweiterung von Bauwerksdatenmodellen erlauben.

Abstract

In the context of this work, the central question is examined how the growing complexity of building information models can be kept manageable. High-order semantic concepts for data exchange in the field of Building Information Modeling and their applications for infrastructure construction will be investigated. The core idea is to exchange information about buildings in the form of object-oriented programs in order to make data models more flexible with regard to their requirements and thus counteract the increasing complexity. A dedicated programming language (IFC-PL) will be developed to describe programs that can be embedded in the existing Industry Foundation Classes (IFC) data model. On the basis of concrete case studies the applicability of the presented approach is presented. It shows how abstract concepts can be described that go beyond the purely traditional data storage and representation of building information models, such as the testing of standards and guidelines. In addition, alternative approaches and concepts are presented that allow a flexible extension of building information models.

Vorwort

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter an der Technischen Universität München am Lehrstuhl für Computergestützte Modellierung und Simulation, an dem ich von Januar 2013 bis März 2018 tätig war.

Mein ganz besonderer Dank gilt Prof. Dr.-Ing. André Borrmann für die Betreuung dieser Arbeit. Darüber hinaus möchte ich mich bei allen Mitarbeitern des Lehrstuhls Computergestützte Modellierung und Simulation sowie des Lehrstuhls Computation in Engineering für die sehr gute Zusammenarbeit bedanken, die fachlich wie auch menschlich immer hervorragend war. Mein Dank gilt auch Herrn Prof. Dr. rer. nat. Rüdiger Westermann für die Übernahme des Zweitgutachtens sowie Prof. Dr.-Ing. Jakob Beetz für die Übernahme des Drittgutachtens. Außerdem möchte ich mich bei allen Koautoren meiner Veröffentlichungen bedanken.

Für das Lektorat der Arbeit danke ich Herrn Josef Huber. Außerdem möchte ich mich bei meinen Eltern Edeltraud und Johann Amann bedanken, die mich auf meinen Weg zur und durch die Promotion stets unterstützt haben.

Die Arbeit widme ich meiner Frau Sophie Amann, die mir stets die nötige Zeit eingeräumt hat, um an meiner Dissertation arbeiten zu können, und mich in allen Phasen der Dissertation unterstützt hat, sowie unserer kleinen Tochter Katharina.

Inhaltsverzeichnis

Abkürzungsverzeichnis

2D	Zweidimensional
3D	Dreidimensional
IFC-PL	IFC Programming Language
API	Application Programming Interface
BASt	Bundesanstalt für Straßenwesen
BIMQL	Building Information Model Query Language
BIM	Building Information Modeling
bSI	buildingSMART International
CAD	Computer Aided Design
CSG	Constructive Solid Geometry
FEM	Finite-Elemente-Methode
Flex	Fast Lexical Analyzer Generator
FLWOR	For, Let, Where, Order by, Return
GDL	Graphic Description Language
GIS	Geoinformationssystem
GML	Geography Markup Language
HTTP	Hypertext Transfer Protocol
IFC	Industry Foundation Classes
ISO	Internationale Organisation für Normung
LLVM	Low Level Virtual Machine
MVD	Model View Definition
NURBS	Non-Uniform Rational B-Spline
OGS	Open Geospatial Consortium
OIP	TUM Open Infra Platform
OKLABI	OKSTRA Klassenbibliothek
OKSTRA	Objektkatalog für das Straßen- und Verkehrswesen
OWL	Web Ontology Language

P21	Part 21
QL4BIM	Query Language for 4D Building Information Models
RAL	Richtlinien für die Anlage von Landstraßen
RAS-L	Richtlinien für die Anlage von Straßen – Teil: Linienführung
RDF	Resource Description Framework
RiZ-ING	Richtzeichnungen für Ingenieurbauten
SDAI	Standard Data Access Interface
SPARQL	SPARQL Protocol And RDF Query Language
STEP	Standard for the Exchange of Product model data
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPath	XML Path Language
XQuery	XML Query Language
XSD	XML Schema Definition Language

Kapitel 1

Einführung

1.1 Motivation

Diese Arbeit untersucht, wie höherwertiges semantisches Wissen in Datenaustauschformate von Bauwerksinformationsmodellen (in dieser Arbeit auch als Bauwerksdatenmodell, Bauwerksmodell oder Building Information Model bezeichnet) in Form von Programmen eingebettet werden kann. Dabei fokussiert sich diese Arbeit auf Bauwerksdatenmodelle für den Infrastrukturbau im Kontext des Ingenieurbauwesens. Innerhalb dieser Arbeit werden nur offene Standards für den Datenaustausch von Bauwerksinformationsmodellen betrachtet, die einen herstellerneutralen Datenaustausch gewährleisten und damit einen Open-BIM-Ansatz ermöglichen, der langfristig auch von der Bundesregierung ab dem Jahr 2020 für Verkehrsinfrastrukturprojekte angestrebt ist (BMVI, 2015). Jedoch sind die hier vorgestellten Konzepte nicht nur auf den Tiefbau beschränkt, sondern können auch auf andere Bereiche wie den Hochbau oder gänzlich andere Domänen wie etwa die Produktdatenmodellierung im Maschinenbau übertragen werden.

Unter höherwertigem semantischem Wissen ist hierbei prozedurales bzw. algorithmisches Wissen gemeint, das in Form von Programmen, die mittels einer Programmiersprache realisiert sind, verpackt und ausgetauscht werden kann. Derzeit in Verwendung befindliche Bauwerksdatenmodelle aus dem Umfeld des Infrastrukturbaus unterstützen beim Datenaustausch im Wesentlichen nur den Austausch von einfachen Attributwerten. Wissen, das in Form von prozeduralen oder algorithmischen Regeln formuliert ist, findet sich derzeit beim Datenaustausch in diesem Kontext nur in sehr eingeschränkter Form wieder. Dabei kann ein Ansatz, der den Austausch von höherwertigem semantischem Wissen unterstützt, helfen, verschiedene Probleme zu vermeiden, die derzeit beim Datenaustausch, basierend auf offenen Standards, auftreten bzw. diese Probleme entsprechend abschwächen. Typische Probleme, die heute Datenaustauschprozesse auf Basis offener Datenformate behindern, sind im Folgenden kurz beschrieben:

– Um überhaupt Daten herstellerneutral zwischen verschiedenen Anwendungen austauschen zu können, muss erst ein gemeinsamer Standard entwickelt

werden, an den sich alle am Datenaustausch beteiligten Partner halten. Solche Standardisierungsvorhaben sind aufwändig und mit einem Kosten- und Zeitaufwand verbunden.

- Definierte Standards müssen erst unterstützt werden. Softwareentwickler müssen entsprechende Import- und Exportfunktionalitäten entwickeln. Dies ist ebenfalls mit einem gewissen Kosten- und Zeitaufwand verbunden. Der Aufwand dafür ist für Softwarehäuser teilweise so hoch, dass diese Standards häufig nur eingeschränkt unterstützt werden. Eine eingeschränkte Unterstützung dieser Standards führt dazu, dass der reibungslose Datenfluss nur begrenzt möglich ist.
- Standards müssen von den Personen, die diese umsetzen müssen, richtig verstanden werden. Hierbei können beispielsweise durch Missinterpretation fehlerhafte Softwareimplementierungen erstellt werden, die sich als weitere Fehlerquellen im Datenaustauschprozess erweisen.
- Die zunehmende Komplexität von Bauwerksdatenmodellen (siehe Abschnitt 2.5) muss handhabbar bleiben. Die Anzahl verschiedener Entitäten und Anwendungsfälle, die durch einen Standard unterstützt werden sollen, nehmen in den etablierten offenen Datenaustauschstandards tendenziell immer weiter zu. Dies macht es zu einer immer größeren Herausforderung für Softwarehäuser, mit diesen Erweiterungen Schritt zu halten.

1.2 Thesen

Um diesen Problemen entgegenzuwirken, werden in dieser Arbeit folgende folgenden grundlegenden Thesen vertreten:

- Um mit der wachsenden Komplexität von Bauwerksmodellen umgehen zu können, müssen Ansätze entwickelt werden, die dieser Entwicklung standhalten können.
- Um langwierige Standardisierungs- und Implementierungsprozesse, die kosten- und zeitaufwändig sind, zu vermeiden, sind Lösungen vorteilhaft, die eine flexible Erweiterung von Bauwerksmodellen zur Laufzeit, also nach ihrer Standardisierung, erlauben, ohne dass eine erneute Standardisierung oder Implementierung dieser Erweiterungen notwendig ist.
- Ein Bauwerksmodell, das flexibel durch Programme erweiterbar ist und dabei Berechnungsvorschriften umsetzt, kann helfen, Fehler zu vermeiden, die bei der Umsetzung auf Basis eines traditionellen Ansatzes gemacht werden können. Bei den heute im Einsatz befindlichen Bauwerksmodellen müssen Softwareentwickler jedes einzelne Attribut eines Datenaustauschstandards selbst interpretieren und, basierend auf dieser Interpretation, selbst Algorithmen implementieren, die diese Attribute verarbeiten. Dabei können Fehler gemacht werden. Durch einen programmbasierten Datenaustausch werden teilweise die attributverarbeitenden Algorithmen mitgeliefert, die eine

Implementierung auf Empfängerseite und die damit verbundenen möglichen Programmierfehler reduzieren.

– Durch einen programmbasierten Datenaustausch wird bei der Definition von Datenaustauschstandards die Standardisierungsarbeit weg von der Spezifikation von einzelnen Attributen auf Objektebene hin zu Schnittstellenvereinbarungen eines Objektes verschoben. Dies ermöglicht es, die Standardisierung auf einem höheren Abstraktionsniveau durchzuführen, bei dem man sich nicht mehr mit einzelnen Details wie Objektattributen befassen muss, sondern auf abstrakte Schnittstellenbeschreibungen fokussieren kann. Dadurch können flexible Standards definiert werden, welche die zunehmende Komplexität beherrschbar machen und die Implementierungskosten reduzieren.

Grundsätzliches Ziel dieser Arbeit ist es, eine Möglichkeit eines programmbasierten Datenaustausches zu beschreiben und anhand von Anwendungsbeispielen die Vorteile dieses Ansatzes gegenüber der derzeit vorherrschenden Philosophie beim Datenaustausch zu beleuchten.

1.3 Aufbau der Arbeit

In Kapitel 2 wird zunächst auf Infrastrukturbauwerke im Kontext des Building Information Modelings eingegangen. Im Anschluss daran werden allgemeine Grundlagen zur Trassierungsplanung mittels Höhen-, Lageplan und Querprofilen beschrieben. Darüber hinaus wird auf STEP (Standard for the Exchange of Product model data) eingegangen und gängige Datenaustauschformate für Trassierungsdaten betrachtet. Das Kapitel schließt mit einem Vorschlag zur Komplexitätsanalyse von Bauwerksdatenmodellen ab.

Das darauf folgende Kapitel (Kapitel 3) betrachtet verwandte Arbeiten mit einer ähnlichen Zielsetzung. Hierbei werden verschiedene Forschungsprojekte, Vorschläge zur Erweiterung von Datenmodellen, bereits existierende und eingesetzte Ansätze sowie produktspezifische Lösungen kritisch untersucht.

Im Kapitel 4 wird der konzeptionelle Ansatz erläutert, der im Rahmen dieser Arbeit gewählt wurde, um die Zielsetzung dieser Arbeit zu realisieren. Dabei werden einige grundlegende Probleme des Datenaustausches von Bauwerksinformationsmodellen beschrieben und ein Konzept vorgestellt, das die Erweiterung von Bauwerksdatenmodellen durch Programme vorsieht. Die Vorteile dieses Konzeptes werden entsprechend erläutert und dargestellt.

Die Kapitel 5, 6 und 7 führen zunächst die Programmiersprache IFC-PL ein, zeigen anhand eines Integrationskonzepts, wie diese in eine Anwendung integriert werden kann, um den Datenaustauschprozess damit zu unterstützten, und gehen auf eine prototypische Implementierung des vorgestellten Ansatzes ein. Im Rahmen der Beschreibung der prototypischen Umsetzung werden ein Early Binding

Generator (oipExpress), eine IFC-PL-Laufzeitumgebung und die TUM Open Infra Platform vorgestellt.

Kapitel 8 behandelt unterschiedliche konkrete Anwendungsfälle. Diese umfassen die Beschreibung von beliebigen Übergangskurven, die parametrische Geometriebeschreibung von Volumenkörpern sowie die Prüfung von Normen und Richtlinien im Kontext des Infrastrukturbaus.

Im vorletzten Kapitel (Kapitel 9) wird auf alternative Ansätze und Konzepte zur IFC-PL eingegangen. Es werden Verfahren, basierend auf XML und Linked-Data, beschrieben.

Das letzte Kapitel endet mit abschließenden Betrachtungen. Dabei werden die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden allgemeine Grundlagen für diese Arbeit eingeführt, die für die folgenden Kapitel vorausgesetzt werden.

2.1 Infrastrukturbauwerke im Kontext des Building Information Modelings

Moderne Verkehrsinfrastruktur setzt sich aus Elementen wie Straßen-, Wasser- und Gleiswegen sowie Brücken- und Tunnelbauten zusammen. Die meisten Infrastrukturbauwerke sind recht unscheinbar, wenn man von monumentalen Bauwerken wie etwa der Golden Gate Bridge absieht. Bei den meisten Verkehrsteilnehmern erregen gewöhnliche Infrastrukturbauwerke keine besondere Aufmerksamkeit. Dabei bewegen wir uns in unserem täglichen Leben sehr intensiv auf diesen Bauwerken. Laut der Studie Mobilität in Deutschland (Lenz *et al.*, 2010) legte im Jahr 2008 jede Person, die in Deutschland lebt, pro Tag im Durschnitt eine Strecke von ca. 39 km zurück. Weiter stellt die Studie fest, dass im Jahr 2008 bei allen Reisezwecken der Pkw das wichtigste Verkehrsmittel war. Daneben nutzten Berufs- und Wochenendpendler in hohem Maß auch die Bahn. Damit spielen Infrastrukturbauwerke eine wichtige Rolle in unserem täglichen Leben.

Mit der zunehmenden Digitalisierung wird die Informatik zum Innovationsmotor technischer und sozialer Fortschritte und gewinnt auch als Enabling-Technologie im Bauwesen stark an Bedeutung. Bauwerke werden hier als Bauwerksmodelle betrachtet, die den gesamten Lebenszyklus von der Planung bis hin zum Rückbau und zur Demontage abbilden. Seit Beginn des Computerzeitalters werden die verschiedenen Lebenszyklusphasen eines Infrastrukturbauwerks durch verschiedene Softwareprodukte- und Standards unterstützt (Amann, 2015). Bereits in den 60er Jahren wurden durch die Organisation *Gemeinsamer Ausschuss Elektronik im Bauwesen* (GAEB) erste Grundlagen für die Digitalisierung von Leistungsverzeichnissen und zur Speicherung von Bestandsdaten entwickelt (Veenhuis, 2017), zu einer Zeit, in der Lochkarten noch als Speichermedien dienten. Auch heute noch ist die Datenart 040 für Trassierungsdaten in ASCII-Form anzutreffen, die

ursprünglich auf Lochkarten gestanzt war (Zwecker, 2014) und vom GAEB entwickelt wurde.

Mit Building Information Modeling (BIM) (Eastman *et al.*, 2008; Borrmann *et al.*, 2015) wurden die technologischen Grundlagen, Werkzeuge und Methoden geschaffen, die die Digitalisierung des Bauens weiter vorantreiben. BIM beschreibt dabei die Idee, alle für den Lebenszyklus relevanten Informationen, die zu einem Bauwerksmodell wie z. B. einem Tunnel, einer Brücke oder Straße gehören, in einem digitalen Bauwerksmodell abzubilden und diese Informationen durchgängig in allen Lebenszyklusphasen zu verwenden. Dabei werden nicht nur Daten zu realen Bauteilen wie beispielsweise Wänden, Pfeilern oder Widerlagern erfasst, sondern darüber hinaus auch abstrakte Objekte wie etwa Prozesse, Akteure oder Zeitpläne. Das Bauwerksmodell selbst wird dabei von unterschiedlichen multidisziplinären Anwendern wie Architekten, Bauingenieuren, Bauherren, Behörden, Versicherungen oder beispielsweise dem Facility Management für ihre jeweiligen Planungs- und Nutzungsziele verwendet und gegebenenfalls spezifisch angepasst und erweitert. Das digitale Bauwerksmodell soll dabei u. a. sicherstellen,

- dass alle relevanten Daten für alle Projektbeteiligten verfügbar sind,
- dass alle Daten in einem konsistenten Zustand sind (die Datenintegrität soll gewährleistet sein),
- dass die Daten effizient genutzt werden können.

Borrmann *et al.* (2015) definieren Building Information Modeling (BIM) wie folgt:

> „Building Information Modeling (BIM) basiert auf der Idee einer durchgängigen Nutzung eines digitalen Gebäudemodells über den gesamten Lebenszyklus eines Bauwerks - vom Entwurf über die Planung und Ausführung bis zum Betrieb des Gebäudes. Sie geht einher mit dem Gedanken eines deutlich verbesserten Datenaustauschs und der dadurch erzielbaren Steigerung der Planungseffizienz durch Wegfall der aufwändigen und fehleranfälligen Wiedereingabe von Informationen."

Dabei beschränkt sich BIM natürlich nicht nur auf Gebäude, sondern kann allgemein auf Bauwerke wie beispielsweise Straßen oder Brücken erweitert werden. BIM ist eine Philosophie, die im Hoch- wie auch im Tiefbau gleichermaßen angewendet werden kann.

Als Building Information Model wird nach (Borrmann *et al.*, 2015) ein „umfassendes digitales Abbild eines Bauwerks mit großer Informationstiefe" verstanden. Für diesen Begriff wird in der Literatur häufiger ebenfalls die Abkürzung BIM verwendet. Im Rahmen dieser Arbeit wird diese Abkürzung jedoch nur als Akronym für Building Information Modeling genutzt und nicht für Building Information

Model. Mit BIM meint man also nicht nur das digitale Bauwerksmodell selbst, sondern auch die damit verbundene Methodik, ein digitales Bauwerksmodell über alle Lebenszyklusphasen hin konsequent zu verwenden. Die konkrete strategische Umsetzung von BIM unterscheidet sich dabei von Unternehmen zu Unternehmen. Eine BIM-Strategie könnte z. B. die automatisierte Erfassung des Baufortschritts (Braun *et al.*, 2014), die automatisierte Überprüfung von Normen und Richtlinien (Eastman *et al.*, 2009; Preidel *et al.*, 2015) oder etwa eine Baustellensimulation zur Aussagengewinnung über projektkritische Zielgrößen (Produktivität, Sicherheit, usw.) (Krepp *et al.*, 2016) beinhalten. Im Kern steht dabei das Bauwerksmodell, jedoch ist das Bauwerksmodell nicht mit BIM gleichzusetzten.

Um digitale Bauwerksmodelle durch computergestützte Werkzeuge verarbeiten zu können, werden Datenmodelle benötigt, welche die Daten beschreiben, mit denen diese Werkzeuge umgehen können. Für den Infrastrukturbereich wurden in den letzten Jahrzenten zahlreiche Datenmodelle vorgeschlagen (Amann, 2015). Diese fanden teilweise auch Einzug in die Praxis. Im Abschnitt 2.4 werden einige gängige Datenaustauschformate aus dem Bereich des Infrastrukturbaus vorgestellt.

2.2 Trassierungsplanung mittels Höhe-, Lageplan und Querprofilen

Die dreidimensionale Lage einer Achse eines Verkehrswegs wird in der Planungsphase im Regelfall mithilfe eines Lage- und Höhenplans beschrieben. Beim Lage- und Höhenplan handelt es sich jeweils um zweidimensionale Pläne. Der Lageplan stellt die Draufsicht auf die Achse und der Höhenplan die Abwicklung der Achse mit entsprechend angetragener Höhe (in y-Richtung) dar. In Abbildung 2.1 ist ein Lageplan dargestellt und Abbildung 2.2 zeigt die entsprechende Abwicklung der Achse mit aufgetragener Höhe.

Im Lageplan wird üblicherweise mit Koordinaten in der Form eines Rechtswerts und Hochwerts gearbeitet. Beispielsweise kann hier mit einem Gauß-Krüger-Koordinatensystem gearbeitet werden, das es ermöglicht, bestimmte Gebiete der Erdoberfläche näherungsweise winkeltreu mithilfe von kartesischen Koordinaten zu verorten. Der Lageplan selbst setzt sich wiederum aus verschiedenen Trassierungselementen zusammen. Diese sind in Abbildung 2.1 mit jeweils unterschiedlichen Farben dargestellt. Typische Trassierungselemente des Lageplans sind Geradenstücke (grün dargestellt), Kreisbögen (rot dargestellt) und Übergangskurven/Übergangsbögen (blau dargestellt). Übergangskurven werden verwendet, um eine Unstetigkeit im Krümmungsverlauf einer Achse zu vermeiden.

Eine Übergangskurve f sowie die Trassierungselemente g und h können, vereinfacht betrachtet, als mathematische Funktionen aufgefasst werden (siehe Abbildung 2.3).

An den Übergangspunkten von g zu f (an der Stelle x_1) und f zu h (an der Stelle x_2) werden üblicherweise eine Reihe von Randbedingungen gestellt:

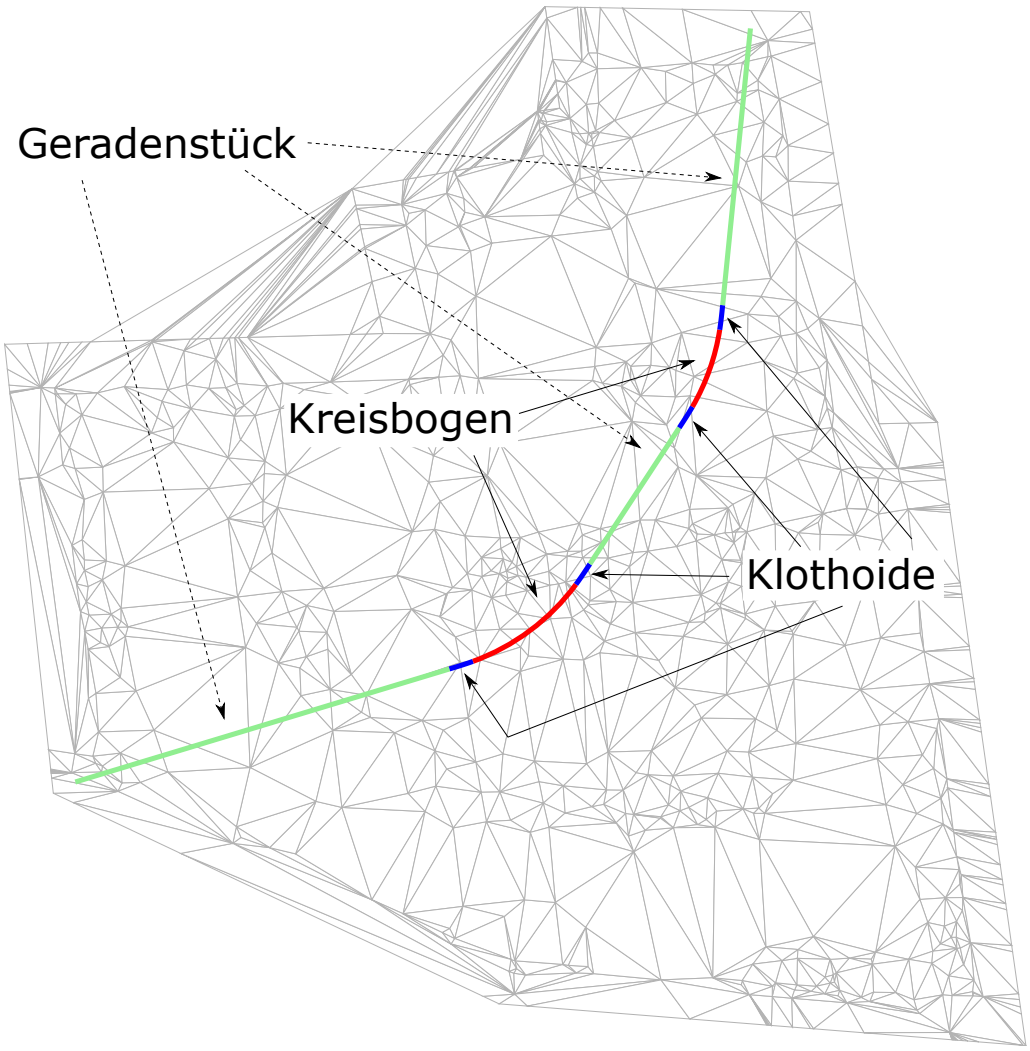


Abbildung 2.1: Darstellung eines Lageplans mit verschiedenen Trassierungselementen

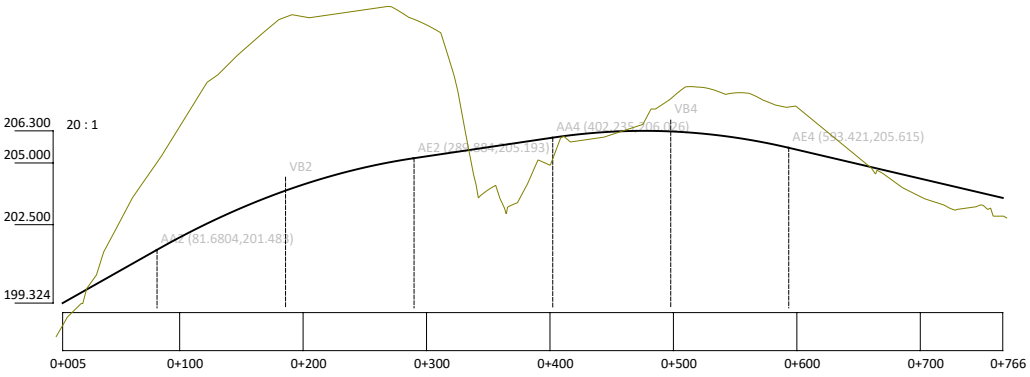


Abbildung 2.2: Darstellung eines Höhenplans, der aus verschiedenen Trassierungselementen besteht. Die ockerfarbene Linie beschreibt die tatsächliche Höhe des Geländemodells an der entsprechenden Stationierung auf der Achse. Die Abbildung ist überhöht dargestellt.

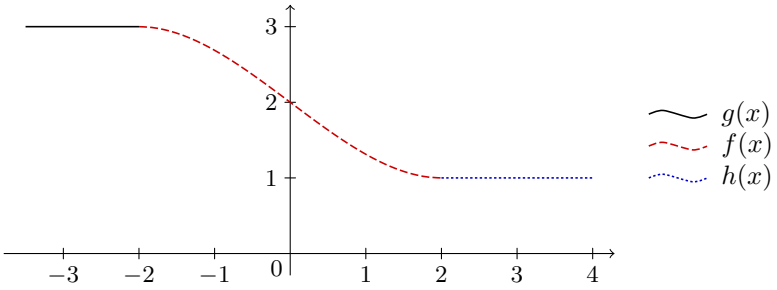


Abbildung 2.3: Eine Übergangskurve f sowie die Trassierungselemente g und h können, vereinfacht betrachtet, als mathematische Funktionen aufgefasst werden.

- Es soll keinen Sprung in der Trassierung geben:
  $g(x_1) = f(x_1)$ und $f(x_2) = h(x_2)$
- Es soll keinen Knick in der Trassierung geben:
  $g'(x_1) = f'(x_1)$ und $f'(x_2) = h'(x_2)$
- Es soll keinen Krümmungsruck geben:
  $g''(x_1) = f''(x_1)$ und $f''(x_2) = h''(x_2)$

Abbildung 2.4 zeigt das Krümmungsband der Achse aus dem Lageplan aus Abbildung 2.1. Auch hierbei wurde wieder die Farbkodierung beibehalten. Das erste Geradenstück (mit der Krümmung 0) geht hier über in einen Übergangsbogen. Dieser besitzt eine konstante Krümmung und stellt die Verbindung zu einem Kreisbogen her. Würde man direkt von einem Geradenstück in einen Kreisbogen übergehen, hätte man eine Unstetigkeit in der Krümmung. Dieser Sprung im Krümmungsband würde sich für einen Autofahrer, der mit hoher Geschwindigkeit fährt, bemerkbar machen, da er plötzlich das Lenkrad stark einschlagen müsste, um die Fahrbahn nicht zu verlassen. Damit ein Autofahrer allmählich mit näherungsweise konstanter Geschwindigkeit das Lenkrad einschlagen kann, werden in der Regel Übergangskurven konstruiert. Im Straßenbau setzt man typischerweise Klothoiden als Überganskurven ein. Die Krümmung einer Klothoide nimmt linear mit ihrer Länge zu.

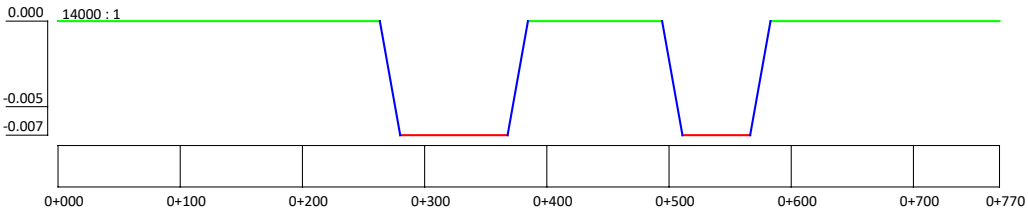


Abbildung 2.4: Darstellung der Krümmung einer Achse im Lageplan. Diese Darstellungsform wird auch als Krümmungsband bezeichnet. Geradenstücke besitzen eine Krümmung mit dem Wert 0 (grün dargestellt). Kreisbögen besitzen einen konstanten Krümmungswert ungleich 0 (rot dargestellt). Klothoiden verändern linear zu ihrer Länge ihre Krümmung (blau dargestellt).

Bei der Fahrt mit einem Schienenfahrzeug muss im Regelfall auch noch die Ableitung des Krümmungsbands stetig sein, da sich andernfalls ein starker seitlicher Ruck bei hoher Fahrgeschwindigkeit bemerkbar machen würde. Im Gleisbau gibt

es eine große Vielzahl von Übergangskurven, die diese Anforderung erfüllen, wie beispielsweise der Wienerbogen oder der Blossbogen. Das Krümmungsband einer Achse, die einige Blossbögen enthält, ist in Abbildung 2.5 dargestellt.

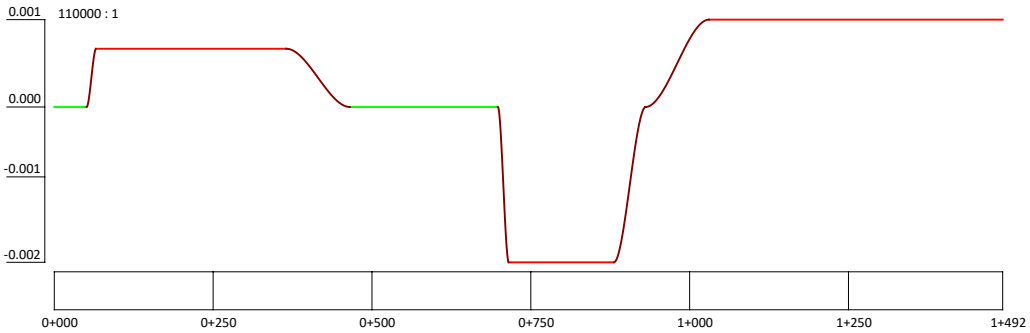


Abbildung 2.5: Krümmungsband einer Achse, die einige Blossbögen enthält. Die Krümmung der Blossbögen ist in dunkelroter Farbe markiert.

Aus dem Lage- und Höhenplan kann der 3D-Trassenverlauf ermittelt werden. Dabei wird der Lageplan mit dem Höhenplan überlagert und daraus die dreidimensionale Achse berechnet (siehe Abbildung 2.6). Die Überlagerung der beiden Pläne erfolgt mittels der Stationierung bzw. der Kilometrierung. Die Stationierung wird linear entlang einer Achse im Lageplan gemessen und ist ebenfalls linear entlang der x-Achse im Höhenplan aufgetragen. Zu einem Startpunkt einer Achse wird im Regelfall eine Startstationierung angegeben, also ein Startwert für die Stationierung. Entlang der Achse im Lageplan können verschiedene Stationierungen, d. h. Längen entlang der Achse, berechnet und mithilfe dieses Wertes (des sogenannten Stationierungswertes) im Höhenplan die entsprechende Höhe ermittelt werden.

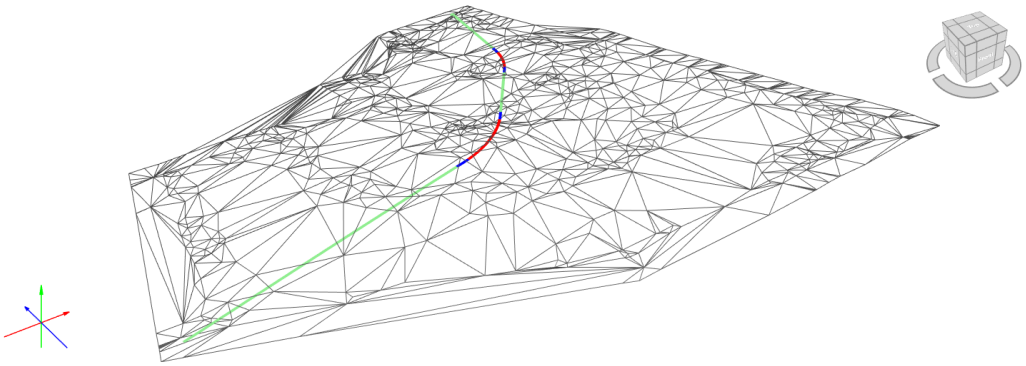


Abbildung 2.6: Darstellung des 3D-Trassenverlaufs auf Basis des Lage- und Höhenplans

Querschnitte einer Trasse können ebenfalls 2D-basiert modelliert werden. Beispiele von Querschnittprofilen sind in Abbildung 2.7 zu sehen.

Querschnittprofile werden mittels einer Stationierung auf der Achse verortet. Die Querprofile werden dabei entweder bereits entsprechend der Fahrbahnquerneigung rotiert modelliert oder es wird ein Standardquerschnitt definiert und zu-

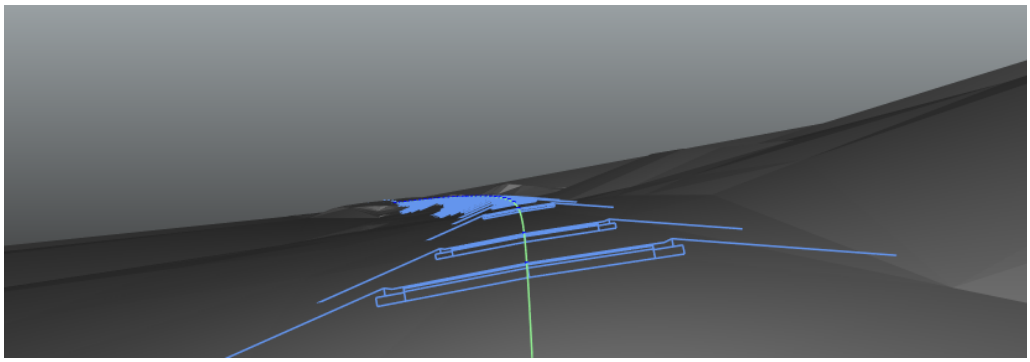


Abbildung 2.7: Verschiedene Querschnittsprofile, die mit dem 3D-Trassenverlauf verortet sind

sätzlich über ein Querneigungsband (siehe Abbildung 2.8) die Verdrehung dieses Profils beschrieben. In diesem Zusammenhang werden oft auch Rampenbänder angegeben, die die Überhöhung des linken bzw. rechten Fahrbahnrands gegenüber der Fahrbahnachse beschreiben.

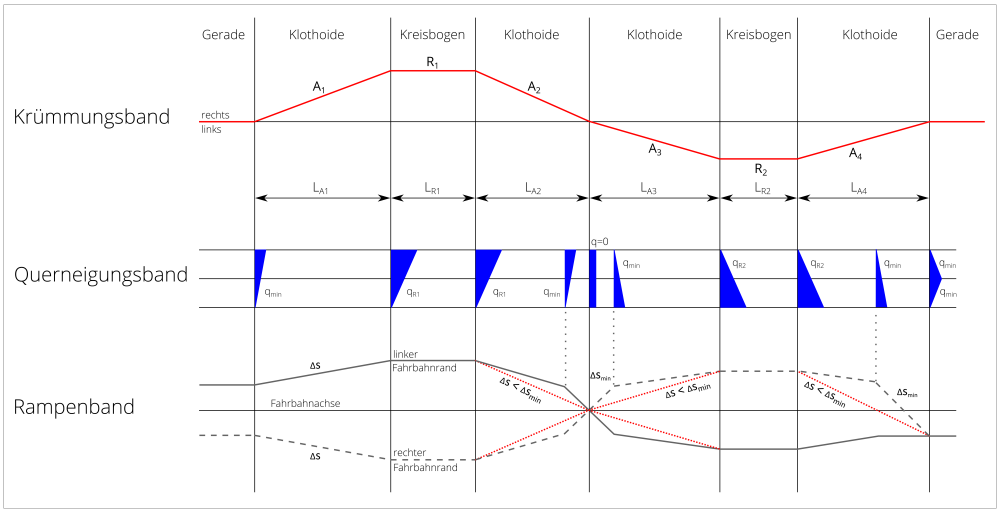


Abbildung 2.8: Beispiel eines Querneigungs- und Rampenbands

Durch das Verbinden der Querprofile entsteht ein 3D-Straßenkörper. In der Abbildung 2.9 ist ein 3D-Straßenkörper mit Einschnitten (rot dargestellt) und Dämmen (grün dargestellt) abgebildet.

Prinzipiell gibt es auch andere Möglichkeiten der Beschreibung von dreidimensionalen Verkehrswegen. Neben dem hier vorgestellten querprofilbasierten Ansatz gibt es beispielsweise einen sogenannte Line-String Ansatz, der unter anderem Verwendung in Finnland findet (Finnish Inframodel) (Hyvärinen, 2011). Dabei werden der rechte bzw. linke Fahrbahnrand oder andere markante Linien wie Bordsteinkanten ebenfalls mithilfe einer Achse, einer sogenannten String-Line, beschrieben. Die Achsbeschreibung selbst kann dabei wieder auf einem Höhe- und Lageplan basieren. Genau so sind aber auch Polylinien, die sich beispiels-

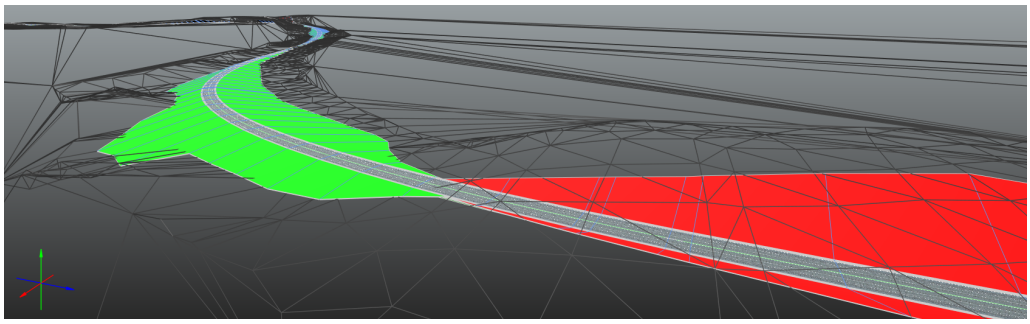


Abbildung 2.9: 3D-Straßenkörper mit Einschnitten (rot dargestellt) und Dämmen (grün dargestellt)

weise aus einfachen Liniensegmenten zusammensetzen, für die Beschreibung von String-Lines zulässig. Querprofile können im String-Line-Ansatz durch die Definition einer Menge von Stationierungswerten auf den unterschiedlichen String-Lines spezifiziert werden.

Grundsätzlich gibt es verschiedene Sichtweisen und Repräsentationen auf einen Straßenkörper: Neben Querprofilen und String Lines werden auch Boundary Representations und Volumenkörper verwendet (Borrmann *et al.*, 2017).

Weitere Details zum Trassierungsentwurf können u. a. in (Hennecke *et al.*, 2000) oder (Richter, 2016) gefunden werden.

2.3 STEP (Standard for the Exchange of Product model data)

STEP (**ST**andard for the **E**xchange of **P**roduct model data) ist ein ISO-Standard (ISO 10303) mit dem Fokus auf den Austausch von Produktdatenmodellen. Diese erfassen sowohl die Geometrie als auch die Semantik von Produkten und den Teilen, aus denen sie bestehen, in einem Datenmodell über alle Lebenszyklusphasen hinweg. STEP hat einen starken Fokus auf den Maschinenbau. Ein Bauwerksmodell kann als Sonderfall eines Produktdatenmodells betrachtet werden, das spezifisch die Details eines Bauwerks abbildet. Der STEP-Standard setzt sich aus verschiedenen Teilen (Parts) zusammen. Im Folgenden wird auf zwei Teilbereiche dieses Standards näher eingegangen:

- Part 11: Description methods: The EXPRESS language reference manual (ISO 10303-11:2004, 2014)
- Part 21: Implementation methods: Clear text encoding of the exchange structure (ISO 10303-21:2016-03, 2016)

2.3.1 Die Datenmodellierungssprache EXPRESS

EXPRESS ist eine objektorientierte Datenmodellierungssprache. Diese wird als Teil 11 (ISO 10303-11:2004, 2014) des ISO-Standards 10303, der in verschiedene

Bestandteile untergliedert ist, spezifiziert. Schenck & Wilson (1994) beschreiben diese wie folgt:

> „EXPRESS is an object-flavored information model specification language which was initially developed in order to enable the writing of formal information models describing mechanical products."

EXPRESS kennt sowohl objektorientierte als auch prozedurale Sprachelemente. Zu den prozeduralen Sprachelementen, die von EXPRESS unterstützt werden, gehören u. a. Variablen, Funktionen, Schleifen und bedingte Anweisungen.

Mithilfe von EXPRESS werden formale Schemata beschrieben, die als zentrales Element Entitäten beinhalten, die Daten von Produkten beschreiben. Ein einfaches Beispiel für ein EXPRESS-Schema ist im folgenden Listing gezeigt:

```
1  SCHEMA Building;
2
3  ENTITY Wall
4      Material: STRING;
5      Length: REAL;
6      Thickness: REAL;
7      Height: REAL:
8  END_ENTITY;
9
10 END_SCHEMA;
```

Ein EXPRESS-Schema wird immer mit dem Schlüsselwort SCHEMA eingeleitet, gefolgt von einem frei zu wählenden Namen (im Beispiel Building). Das Schema kann eine Reihe von Entitäten beinhalten. Im Beispiel enthält dieses nur eine Entität mit dem Namen Wall. Entitäten sind in etwa vergleichbar mit dem Konzept einer Klasse in der objektorientierten Programmierung. Eine Entität beschreibt keine konkrete Instanz, sondern nur die Struktur einer möglichen Instanz (Ausprägung) des Entitätstyps.

Eine Entität kann Attribute besitzen. Die Entität Wall beinhaltet vier Attribute (Material, Length, Thickness, Height). Ein Attribut hat neben einem Namen auch einen Attributtypen. EXPRESS kennt vordefinierte Attributtypen wie z. B. INTEGER, REAL oder STRING. Zudem ist es möglich, eigene Typen auf Basis bereits vorhandener EXPRESS-Typen zu definieren. Eine einfache EXPRESS-Typdefinition ist im folgenden Listing dargestellt:

```
1  TYPE IfcAreaMeasure = REAL;
2  END_TYPE;
```

Diese beschreibt einen Typ mit dem Namen (IfcAreaMeasure), der für Flächeninhalte genutzt werden kann.

Die Definition einer Entität gliedert sich im Allgemeinen in folgende Abschnitte, die allesamt optional sind:

- Explizite Attribute: Beschreiben Attribute der Entität mittels Attributnamen und Attributtypen. Attribute können auch als optional definiert werden.
- Derive Clauses: Werden auch als Derived-Attribute bezeichnet. Diese Attribute werden nicht explizit gespeichert, sondern können aus den bestehenden Attributen hergeleitet werden.
- Inverse Clauses: Modelliert inverse Relationen.
- Unique Clauses: Definiert Attribute, deren Attributwerte nicht mehrmals im Datenbestand vorkommen dürfen.
- Where Clauses: Werden durch Where-Regeln realisiert. Diese werden zur Realisierung von Einschränkungen des Wertebereichs einer Entität genutzt.

Derived-Attributes sind Attribute, die aus den vorhandenen Daten einer Entität abgeleitet werden können. Dabei definiert ein Derived-Attribute mittels der EXPRESS-Sprache formal, wie dieses zu berechnen ist. Folgendes Listing zeigt exemplarisch das Attribut Länge (length), das aus den Attributen start und end berechnet wird[1]:

```
1  ENTITY MyLine;
2      start : point;
3      end : point;
4  DERIVE
5      length : distance:=SQRT((end.x − start.x))**2
6                                  + (end.y − start.y)**2);
7  END_ENTITY;
```

Where-Regeln sind auch auf der Ebene der EXPRESS-Typen zu finden. Folgendes Beispiel zeigt eine Where-Regel für einen EXPRESS-Typ, die überprüft, ob sich ein Wochentag innerhalb des validen Wertebereichs befindet:

```
1  TYPE IfcDayInWeekNumber = INTEGER;
2   WHERE
3    ValidRange : {1 <= SELF <= 7};
4  END_TYPE;
```

Folgendes Beispiel zeigt eine Where-Regel innerhalb einer Entität[2]:

```
1  ENTITY IfcActorRole;
2    Role : IfcRoleEnum;
3    UserDefinedRole : OPTIONAL IfcLabel;
4    Description : OPTIONAL IfcText;
5   INVERSE
6    HasExternalReference : SET [0:?] OF IfcExternalReferenceRelationship FOR
7    RelatedResourceObjects;
8   WHERE
9    WR1 :  (Role <> IfcRoleEnum.USERDEFINED) OR
10        ((Role = IfcRoleEnum.USERDEFINED) AND
11          EXISTS(SELF.UserDefinedRole));
12 END_ENTITY;
```

[1] Der Doppelsternoperator ** wird zum Potenzieren verwendet.
[2] <> ist hierbei der Not-Equal-Operator

Durch die Where-Regel im vorangegangenen Beispiel wird folgender Sachverhalt abgebildet: Eine Instanz der Entität IfcActorRole muss, wenn die Rolle (Role) mit dem Wert USERDEFINED belegt ist, das Attribut UserDefinedRole bereitstellen.

EXPRESS erlaubt auch die Definition von Regeln (Rules). Der typische Aufbau einer solchen Regel ist im Folgenden dargestellt:

```
1  RULE rule_name FOR (entity_type_1, ..., entity_type_N);
2      (* executable statements *)
3  WHERE
4      (* some expression that returns TRUE or FALSE *)
5  END_RULE;
```

Eine Regel, die für eine Entität (aircraft) überprüft, ob die Anzahl der gebuchten Plätze nicht die Anzahl aller existierenden Plätze überschreitet, ist im folgenden Listing dargestellt:

```
1  RULE max_number_of_passengers FOR (aircraft);
2  WHERE
3     max_is_853 : SIZEOF(bookshelf) <= 853;
4  END_RULE;
```

Damit eine Where- oder Derived Clause bzw. eine Rule nicht zu unübersichtlich wird, definiert EXPRESS Funktionen. Diese ermöglichen es, bestimmte abgeschlossene Funktionalitäten wiederverwenden zu können. Der typische Aufbau einer Funktion wird im Folgenden gezeigt:

```
1  FUNCTION functionName(
2        parameterName1 : entityType1, ..) : resultType;
3  LOCAL
4      localVariableName : variableType;
5       ...
6  END_LOCAL;
7      REPEAT i := 1 to SIZEOF(localVariableName);
8          IF (...)  THEN
9              result  := result  + 1;
10         END_IF;
11     END_REPEAT;
12 RETURN(result);
13 END_FUNCTION;
```

Funktionen können Eingabeparameter besitzen. Außerdem können diese Variablen definieren und auf Basis von prozeduralen Anweisungen wie Schleifen oder Bedingungen Berechnungen durchführen. Die berechneten Werte können mittels einer Return-Anweisung zurückgegeben werden.

Zwischen verschiedenen Entitäten können Vererbungsbeziehungen existieren. Dies ist ein objektorientiertes Sprachkonzept, welches durch EXPRESS realisiert wird. Eine Entität kann auch als abstrakte Entität (abstrakte Klasse) definiert werden.

Für weitere Details zum Sprachumfang von EXPRESS sei hier auf entsprechende Literatur wie z. B. (Schenck & Wilson, 1994) verwiesen. Die vollständige Grammatik der EXPRESS-Sprache befindet sich im Anhang C.1.

2.3.2 STEP-Clear-Text-Encoding

Um Daten, die auf Basis eines EXPRESS-Schemas definiert sind, austauschen zu können, wurde das STEP-Clear-Text-Encoding eingeführt. Dieses ist als Part 21 von STEP (ISO 10303-21:2016-03, 2016) standardisiert und stellt eine textuelle Beschreibung von STEP-Daten dar, die menschenlesbar ist. Das Format ist auf Instanzebene angesiedelt, das heißt, es beschreibt nicht wie EXPRESS auf Schemaebene den allgemeinen Aufbau von produktspezifischen Daten, sondern ganz konkret einzelne Ausprägungen (Instanzen) von Entitäten.

Ein einfaches Beispiel für ein STEP-Clear-Text-Encoding, das auf dem zuvor gezeigten EXPRESS-Schema Building basiert, wird im folgenden Listing gezeigt:

```
1  ISO−10303−21;
2  HEADER;
3  FILE_DESCRIPTION((’Building’),’2;1’);
4  FILE_NAME(’test.stp’,’2016−04−20T11:30:17’,
5  (’’),(’’,’’),’’,’WALL’,’’);
6  FILE_SCHEMA((’Building’));
7  ENDSEC;
8  DATA;
9  #1=WALL(’Concrete’, 10.22, 0.23, 2.34);
10 ENDSEC;
11 END−ISO−10303−21;
```

Der genaue Aufbau dieses Formats ist ebenfalls mittels einer Grammatik beschrieben, die in (ISO 10303-21:2016-03, 2016) zu finden ist. Die tatsächlichen Daten der Instanzen von Entitäten befinden sich in der sogenannten Data-Section. Diese wird durch den Text DATA in Zeile 8 eingeleitet. Darüber befindet sich die sogenannte Header-Section, die Angaben zur Struktur der Daten, beispielsweise den Namen des zugrundeliegenden EXPRESS-Schemas, enthält. Eine Sektion (Section) wird durch den Text ENDSEC abgeschlossen. In Zeile 9 befindet sich eine Instanz der Entität Wall. In runden Klammern folgen die Werte für die Attribute, die durch das Schema (Building) definiert werden (Material, Length, Thickness und Height). Jede Instanz einer Entität wird mit einer eindeutigen Nummer versehen. Die Instanz im Beispiel hat die Nummer #1. Diese Nummern werden verwendet, um Assoziationen zwischen verschiedenen Entitäten beschreiben zu können. Dies ist der Fall, wenn eine Entität ein Attribut besitzt, das als Typ eine andere Entität besitzt. Das Clear-Text-Encoding wird in der Literatur auch als STEP-Physical File bezeichnet. Weitere Details zu diesem Format können u. a. in (Anderl, 2000) oder (Borrmann *et al.*, 2015) gefunden werden.

2.3.3 Early- und Late-Binding

Zum Lesen und Schreiben von STEP-Dateien mit dem Clear-Text-Encoding gibt es zwei etablierte Ansätze. Diese werden als Early- und Late-Binding-Ansatz unterschieden (Amann *et al.*, 2015).

Bei einem Early-Binding-Ansatz wird jede Entität eines EXPRESS-Schemas auf eine Entität (Klasse) der Zielsprache (Programmiersprache, in der die STEP-Daten verarbeitet werden sollen) abgebildet. Zudem werden Funktionen bereitgestellt, die das Lesen und Schreiben von STEP-P21-Dateien auf Basis dieses Mappings ermöglichen. Beim Lesen einer STEP-P21-Datei wird jede STEP-Instanz in eine entsprechende Instanz (Objekt) der Zielsprache konvertiert. Umgekehrt können beim Schreiben Objekte der Zielsprache, deren Entitätstypen (Klassentypen) aus dem EXPRESS-Schema abgeleitet wurden, wieder in STEP-P21-Datei umgewandelt werden. Die Bindung der Entitäten eines EXPRESS-Schemas an eine Zielsprache folgt hierbei zur Übersetzungszeit (Compile-Zeit).

Im Gegensatz zu diesem Ansatz folgt beim Late-Binding-Ansatz die Bindung von Instanzen der STEP-P21-Datei zu Instanzen der Zielsprache erst zur Laufzeit. Dazu wird eine standardisierte API bereitgestellt, die den Namen SDAI (Standard Data Access Interface) trägt. Ein Language-Binding dieser sprachunabhängig formulierten API wird im STEP-Standard in den Teilen 23, 24 und 27 für die Programmiersprachen C++, C und Java beschrieben. Die API des SDAI erwartet Namen von Entitäten und Attributen als String-Werte, die erst zur Laufzeit ausgewertet werden. Sollte ein Attribut oder eine Entität nicht existieren, die mittels der SDAI verwendet werden, erhält man diesen Fehler erst zu Ausführungszeit eines Programmes. Bei einem Early-Binding Ansatz erhält man derartige Fehler schon während der Übersetzungszeit des Programms.

Weitere Details zur Umsetzung und zu den Vor- und Nachteilen beider Ansätze finden sich u. a. in (Anderl, 2000) und (Amann *et al.*, 2015).

2.4 Gängige Datenaustauschformate für Trassierungsdaten

Im folgenden Abschnitt werden einige gängige Datenaustauschformate aus dem Bereich des Infrastrukturbaus mit Fokus auf Austauschdatenformate für Trassierungsdaten beschrieben.

2.4.1 Das Datenmodell IFC 4.1

IFC (Industry Foundation Classes) ist ein standardisiertes Datenformat, das von der Organisation buildingSMART International für den Austausch von Bauwerksmodellen entwickelt wurde (Laakso & Kiviniemi, 2012). Bis zur Version 4.0 war der Fokus von IFC stark auf den Hochbau ausgerichtet. Mit der aktuellen Entwicklung IFC 4.1 werden jetzt tief- und infrastrukturbaurelevante Anwendungsfälle besser durch die IFC unterstützt (Borrmann *et al.*, 2015). Eine der größten

Neuerungen der Version 4.1 gegenüber der Version 4.0 ist die Erweiterung durch Trassierungselemente.

Die Erweiterungen von IFC 4.1 basieren auf einem konzeptionellen Modell für Trassierungen (Amann *et al.*, 2014), das in Zusammenarbeit zwischen buildingSMART und der OGC (OpenGIS Consortium) entwickelt wurde (Amann & Borrmann, 2015a). Auf Seiten der OGC wurde dieses konzeptionelle Modell durch den Standard InfraGML realisiert. Dieser Ansatz wurde gewählt, um die Harmonisierung zwischen der BIM- und GIS-Welt zu unterstützen. InfraGML und IFC 4.1 besitzen hierbei in ihren jeweiligen Datenmodellen eine gewisse Überlappung, unterstützen aber darüber hinaus jeweils noch weitere Anwendungsfälle, die für die jeweilige Domäne spezifisch sind (Amann & Borrmann, 2015a). So ist im Rahmen von InfraGML beispielsweise auch die Unterstützung des Datenaustauchprozesses im Bereich der Landvermessung und der Liegenschaftsverwaltung vorgesehen, was jedoch außerhalb des Anwendungsbereichs der IFC liegt. Im Gegensatz dazu spielen detaillierte Beschreibungsmöglichkeiten von Teilbauwerken wie Wänden in InfraGML keine Rolle, werden jedoch seitens der IFC 4.1 unterstützt.

Abbildung 2.10 zeigt einen Überblick über die neu in das IFC 4.0 Schema eingeführten Entitäten. IFC 4.1 unterstützt Lage- und Höhenpläne. Die Entität IfcAlignment2DHorizontal wird genutzt, um einen Lageplan zu beschreiben. Diese besteht aus einer beliebigen Anzahl von Segmenten (Segments), die auf die einzelnen Trassierungselemente des Lageplans verweisen. Ein Lageplan setzt sich dabei aus Geradenstücken (IfcLineSegment2D), Kreisbögen (IfcCircularArcSegment2D) und Übergangskurven (IfcClothoidalArcSegment2D) zusammen. Der Höhenplan wird durch die Entität IfcAlignment2DVertical beschrieben. Dieser besitzt Geradenstücke (IfcAlignment2DVerSegLine), Kreisbögen (IfcAlignment2DVerSegParabolicArc) und Parabeln (IfcAlignment2DVerSegCircularArc).

Das gebräuchlichste Format zur Speicherung von IFC-Dateien stellt das STEP-Format (siehe Abschnitt 2.3.2) dar. Es gibt zwar auch eine XML-Variante der IFC-Instanz-Dateien, die jedoch aus historischen Gründen derzeit nicht so weit verbreitet ist wie das STEP-äquivalent.

Weitere Details zum IFC 4.1-Datenmodell sind u. a. in (buildingSMART, 2018c), (Amann & Borrmann, 2015c) und (Amann & Borrmann, 2015a) zu finden.

2.4.2 Das OKSTRA-Datenmodell

OKSTRA ist die Abkürzung für **O**bjekt**k**atalog für das **Stra**ßen- und Verkehrswesen. Die derzeit aktuelle Version von OKSTRA ist die Version 2.018 (Bundesanstalt für Straßenwesen, 2018). Ursprünglich wurde OKSTRA als Forschungsprojekt der Bundesanstalt für Straßenwesen (BASt) gestartet und im Jahr 2000 vom damaligen Bundesministerium für Verkehr, Bau und Stadtentwicklung als bundesweiter Standard für den Bereich der Bundesfernstraßen eingeführt (Bundesministerium für Verkehr, Bau- und Wohnungswesen, 2010; Bundesanstalt für Straßenwesen, 2018). Der Standard ist in eine Reihe verschiedener Schemata un-

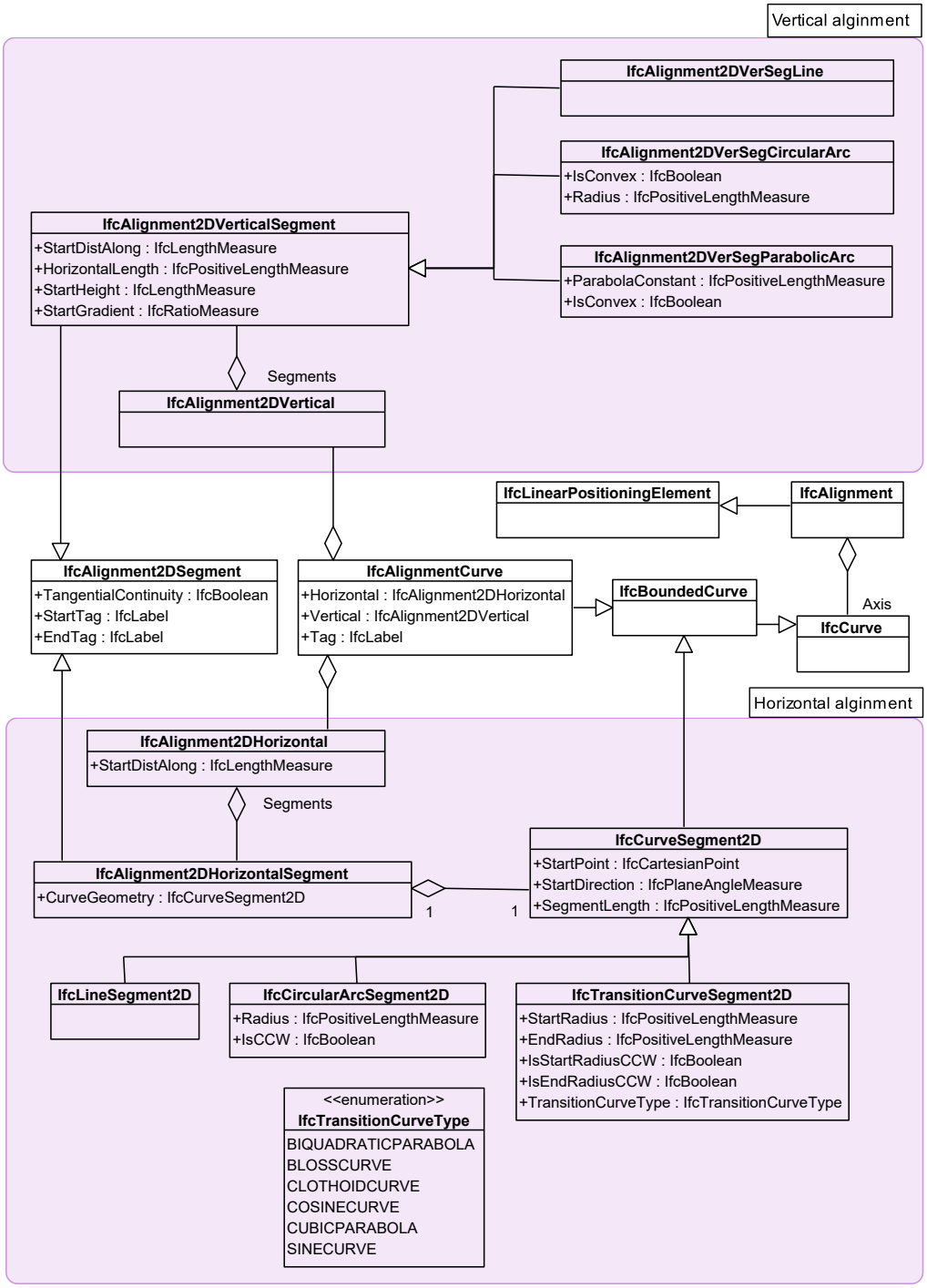


Abbildung 2.10: UML-Klassendiagramm, das einen Überblick über die wichtigsten trassierungsrelevanten Entitäten aus IFC 4.1 zeigt

tergliedert. Diese umfassen Bereiche wie Verkehrsdaten, Zustandsdaten, Umweltdaten, Unfalldaten oder auch Entwurfsdaten. Die trassierungsrelevanten Informationen stecken im OKSTRA-Entwurfsschema. Eine Übersicht über die Datenmodellierung einer Trasse (Alignment) ist in Abbildung 2.11 dargestellt.

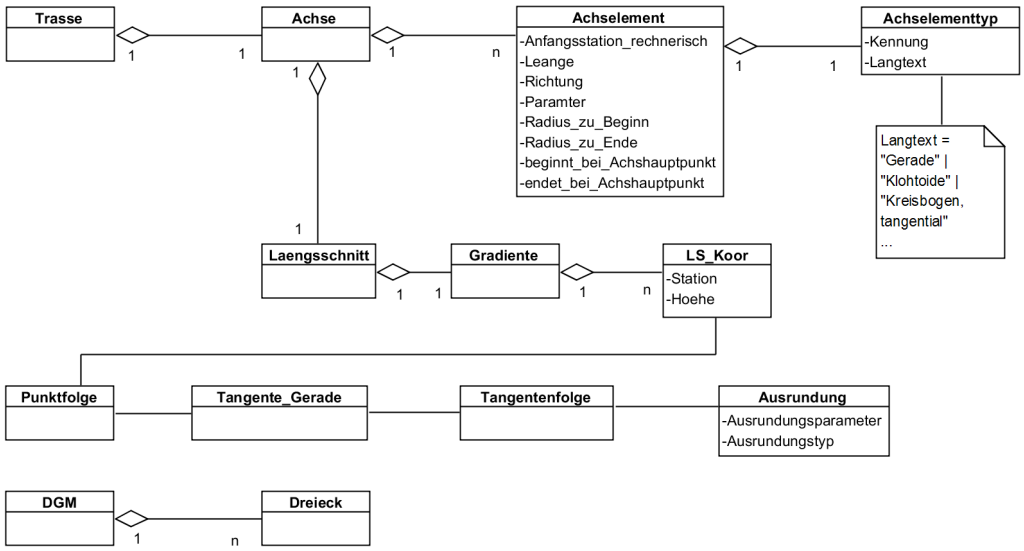


Abbildung 2.11: OKSTRA-Datenmodell für Trassen

Eine Trasse wird durch die Klasse Trasse beschrieben. Eine Trasse besteht aus einer Achse, diese wiederum aus Achselementen (Achselement). Ein Achselement ist dabei ein Trassierungselement eines Lageplans. Dies können Geradenstücke, Kreisbögen und Klothoiden sein. Der Typ eines Achselements wird dabei über ein Attribut, das den Achselementtyp speichert, festgelegt (Amann & Borrmann, 2015a). Neben dem Lageplan besitzt die Achse auch einen Höhenplan. Dieser wird in OKSTRA visierpunktbasiert gespeichert (Amann & Borrmann, 2015a). Abbildung 2.12 zeigt den Unterschied zwischen einem segmentbasierten und einem visierpunktbasierten Höhenplan.

Beim segmentbasierten Ansatz werden die Informationen der einzelnen Segmente gespeichert. Beispielsweise werden für jede Gerade und jede Parabelausrundung der Start- und Endpunkt des entsprechenden Segments gespeichert. Beim visierpunktbasierten Ansatz werden nur die Visierpunkte und die Ausrundungsradien gespeichert. Die Start- und Endpunkte der einzelnen Segmente müssen hier explizit berechnet werden. Beide Varianten können ineinander umgerechnet werden. Weitere Details zum Aufbau des Höhenplans können in (Amann & Borrmann, 2015a) nachgelesen werden.

2.4.3 LandXML

LandXML basiert auf XML und besitzt ähnlich wie IFC 4.1 und OKSTRA Elemente zur Beschreibung von Trassierungselementen. Der Höhenplan in LandXML ist visierpunktbasiert. LandXML war lange Zeit der De-facto-Standard bei der Übergabe von Trassierungsinformationen und wird auch heute noch zu diesem

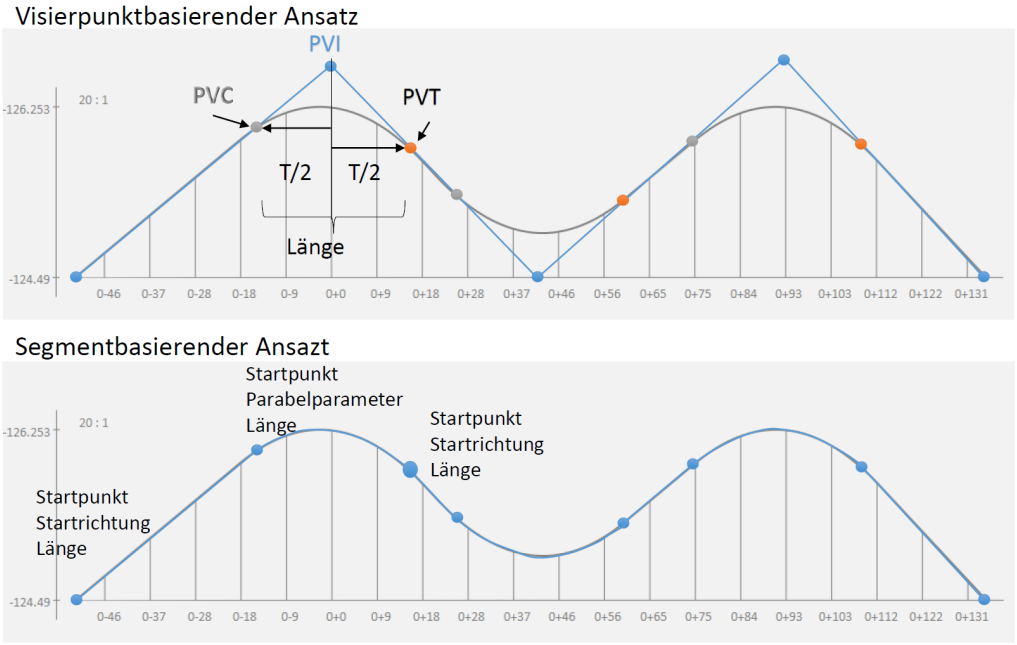


Abbildung 2.12: Visierpunkt- und segmentbasierender Ansatz im Vergleich

Zweck genutzt (Rebolj *et al.*, 2008; Ziering *et al.*, 2007). Neben Trassierungsinformationen können damit auch digitale Geländemodelle und Querprofile ausgetauscht werden. Zudem bietet LandXML eine grundlegende Unterstützung für die digitale Beschreibung der Liegenschaftsverwaltung und für Vermessungsdaten wie z. B. Punktewolken. Die erste Version von LandXML, die eine große Verbreitung und Unterstützung fand, war die Version 1.0. Diese wurde im Jahr 2000 veröffentlicht. Im Jahr 2006 folgte dann die Version 1.1 und im Jahr 2008 die Version 1.2 (LandXML.org, 2018). Version 1.2 wird von vielen Softwarewerkzeugen unterstützt und besitzt momentan unter den LandXML-Versionen die größte Verbreitung. 2014 wurde die LandXML Version 2.0 veröffentlich, die sich aber einer geringeren Beliebtheit erfreut. Ein Grund hierfür ist u. a. die fehlende bzw. mangelhafte Organisation hinter dem LandXML-Standard. 2013 startete deshalb die LandInfra-Initiative, die von der OGC initiiert wurde, den Versuch, sich dieses Problems anzunehmen und LandXML unter dem Dach der OGC weiterzuentwickeln. In einem ersten Schritt wurde dabei der aktuelle Stand von LandXML (damals Version 1.2) untersucht. Als Endergebnis wurde allerdings beschlossen, LandXML nicht weiterzuentwickeln, sondern stattdessen eine Neuentwicklung namens InfraGML (ursprünglich unter dem Namen LandInfra GML geführt) zu forcieren, die besser in das Umfeld der OGC-Standards passt und einige Designfehler von LandXML beseitigt (Scarponcini, 2013).

2.5 Bewertung der Komplexität von Bauwerksdatenmodellen

Anhand der Weiterentwicklung offener Standards und Spezifikationen über die letzten Jahre für die Arbeitsmethode BIM, beispielsweise der Industry Foundation Classes oder des OKSTRA-Standards, ist eine zunehmende Komplexität dieser Standards für Bauwerksdatenmodelle beobachtbar. Beispielsweise hat die Zahl der Entitäten von ca. 180 Entitäten im IFC 1.5.0 Standard auf über 750 Entitäten in der Version 4.0 zugenommen. In ähnlicher Weise hat sich die Anzahl der Attribute der verschiedenen Entitäten entwickelt. Von ursprünglich ca. 150 Attributen im IFC 1.5.0 Standard ist hier ein Anstieg auf über 1400 Attribute im IFC 4.0 Standard verzeichenbar. Die Anzahl der definierten Typen hat ebenfalls zugenommen. IFC 1.5.0 besitzt ca. 100 verschiedene Typen, IFC 4.0 dagegen schon fast 400 verschiedene Typen. Auch anhand anderer Zahlen wie der durchschnittlichen Tiefe des Vererbungsbaums aller Entitäten oder der Anzahl eingeführter Propety-Sets oder der Anzahl von Function-, Rules- and Where-Klauseln ist eine zunehmende Komplexität des IFC-Standards erkennbar. In (Amor, 2015) und (Amor *et al.*, 2007) wurde die Evolution des IFC-Schemas mit verschiedenen Metriken aus dem Bereich des Software Engineerings untersucht. Die Autoren stellen fest, dass das IFC-Schema mit Version 4.0 stark angewachsen ist; allerdings sei die Komplexität an einigen Stellen auch verringert worden (Anzahl der Basisklassen, Erhöhung optionaler Attribute, usw.). Trotzdem ist nach den Autoren insgesamt eine steigende Komplexität zu beobachten.

Eine ähnliche Entwicklung zeichnet sich auch beim OKSTRA-Standard ab. Die Anzahl der Klassen, gemessen an der Anzahl der komplexen Typen (Complex Types) im XML-Schema, hat stetig zugenommen (siehe Abbildung 2.13). Die Anzahl der Attribute ist von der Version 1.008 (veröffentlich am 09.09.2003) bis zur Version 2.016 (veröffentlicht am 17.02.2014) stetig angewachsen, mit einer einzigen Ausnahme. Kurzzeitig ging die durchschnittliche Anzahl von Attributen je Klasse (in der XML-Terminologie auch als Complex Type bezeichent), gemessen anhand der durchschnittlichen Anzahl von Attributen je Klasse, zurück, stieg jedoch mit den letzten neueren Versionsveröffentlichungen stark an, von ca. zwei Attributen auf fast fünf Attribute je Klasse.

Auch bei dem Quasistandard LandXML ist ein Anstieg bei den Anzahl der Klassen, Aufzählungstypen und der Anzahl der Attribute sichtbar (siehe Abbildung 2.13).

Mit der zunehmenden Komplexität von BIM-Modellen wird es auch für Softwareanbieter immer schwieriger, mit diesen technischen Weiterentwicklungen mitzuhalten. Eine zentrale Fragestellung hierbei ist, wie man mit der wachsendenden Komplexität von Building Information Models umgehen kann und welche Ansätze es gibt, die Datenintegrität und einfache Erweiterbarkeit und die damit einhergehende Komplexitätssteigerung sinnvoll zu managen.

Im Bereich des Software-Engineering können Metriken wie die Tiefe der Vererbungshierarchie, die Anzahl der Kinder einer Klasse oder die Kopplung zwischen

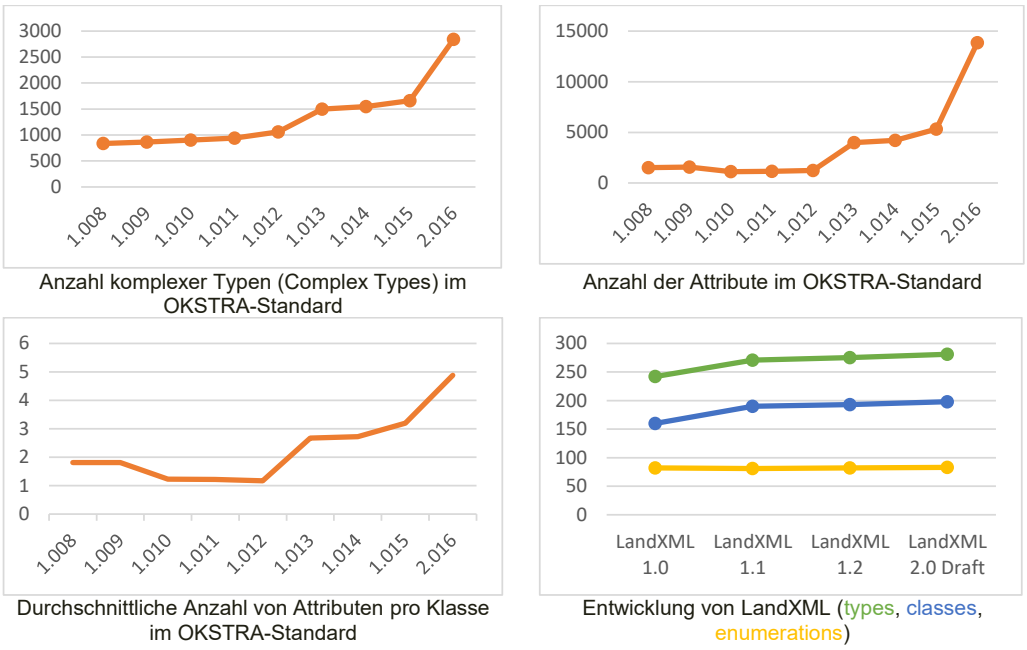


Abbildung 2.13: Steigerung der Komplexität vom BIM-Modellen am Beispiel von OKSTRA und LandXML

verschiedenen Klassen verwendet werden, um eine Vorstellung davon zu erhalten, wie komplex ein Datenmodell ist (McConnell, 2004). Aber obwohl diese Metriken einen Hinweis darauf geben, wie sich ein Schema entwickelt hat, ist es nicht geeignet, verschiedene Bauwerksdatenmodelle miteinander zu vergleichen. Vergleicht man beispielsweise OKSTRA 2.016 mit LandXML 1.2 und IFC 4.1, indem man nur die Anzahl der Klassen in jedem Standard zählt, zeigt sich, dass OKSTRA 2.016 viermal komplexer als IFC 4.1 und zehnmal komplexer als LandXML 1.2 ist. Dieser Vergleich ist jedoch nicht wirklich sinnvoll, da OKSTRA viel mehr Anwendungsfälle abdeckt als LandXML oder IFC 4.1.

Um eine bessere Metrik für den Vergleich zwischen verschiedenen BIM-Modellen zu erhalten, macht es Sinn, eine Metrik auf Instanzebene anzuwenden (Amann & Borrmann, 2016). Anstatt Klassen auf Schemaebene zu zählen, ist es sinnvoller, Klassen, die auf Instanzebene verwendet werden, zu zählen. Um diese Metrik anzuwenden, wird mindestens eine Instanzdatei benötigt, die als Testfall- bzw. Anwendungsfallszenario dient. Der in Abbildung 2.14 dargestellte Testfall soll nun anhand dieser Metrik bewertet werden.

Das Testszenario aus Abbildung 2.14 wurde in den Dateiformaten IFC 4.1, OKSTRA 2.016 und LandXML 1.2 gespeichert. Alle drei Dateien enthalten die gleichen Daten: ein digitales Geländemodell und mehrere Alignments (Trassierungen), die jeweils einen Lage- und einen Höhenplan umfassen. Der Lageplan setzt sich aus Geradenstücken, Kreisbögen und Klothoiden zusammen, der Höhenplan aus Parabelstücken und Geradenstücken. Aus jeder Datei kann jede andere Datei

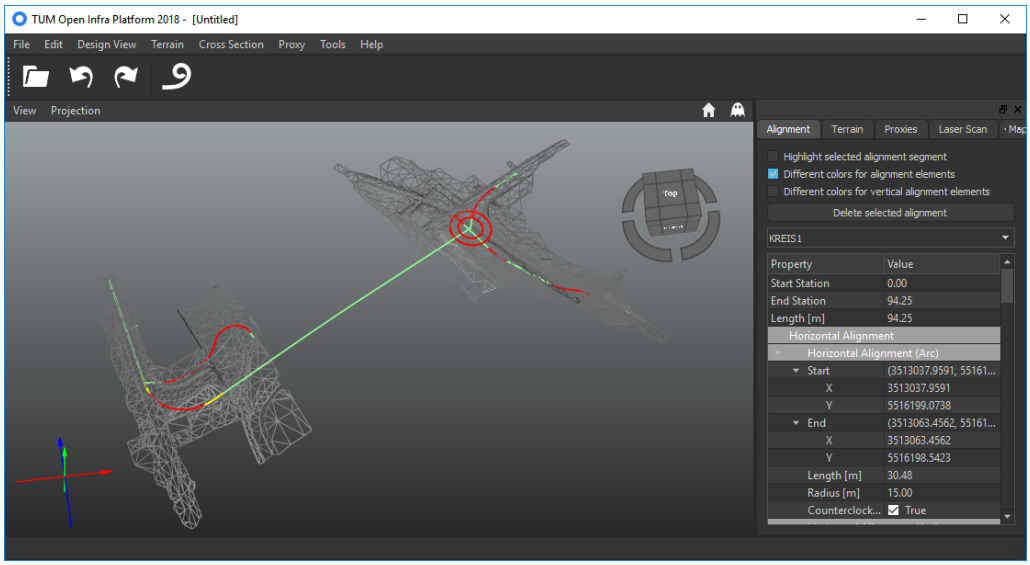


Abbildung 2.14: Testfall: Verschiedene Trassierungen mit einem digitalen Höhenmodell

erzeugt werden[3]. Für die Konvertierung von einem Dateiformat in ein anderes wurde die Software TUM Open Infra Platform (Amann *et al.*, 2016) verwendet. Die Metrik zählt alle unterschiedlichen Vorkommen von Tags in den XML-basierten Instanzdokumenten. In STEP-P21-Dateien werden alle unterschiedlichen Vorkommen von Entity-Namen gezählt. Tabelle 2.1 zeigt die berechneten Ergebnisse.

BIM-Modell	**Testfall 1**
LandXML 1.2	23 (unterschiedliche Vorkommen von Tags)
IFC 4.1	32 (unterschiedliche Vorkommen von Entitätsnamen)
OKSTRA 2.016	42 (unterschiedliche Vorkommen von Tags)

Tabelle 2.1: Anzahl unterschiedlicher Vorkommen von Tags und Entity-Namen

Im untersuchten Datenaustauschszenario (Testfall 1) stellt sich heraus, dass OKSTRA nicht zehnmal so viele Entitäten wie beispielsweise LandXML benötigt, um den Datenaustausch zu realisieren, sondern nur etwa um den Faktor zwei mehr Entitäten. Diese Metrik liefert damit eine bessere Grundlage für den Vergleich verschiedener Bauwerksmodelle. Laut der Metrik ist der OKSTRA-Datenaustauch etwas komplexer als der LandXML-basierte Datenaustauch, jedoch nur um etwa den Faktor 2 und nicht um den Faktor 10. Mit der hier beschriebenen Metrik, die auf Instanzebene misst, wird die Komplexität eines BIM-Modells durch die Anzahl der in einer Instanzdatei verwendeten Entitäten definiert. Die Metrik könnte theoretisch noch weiter ausgebaut werden und außerdem beispielsweise die Anzahl unterschiedlich verwendeter Attributwerte miteinbeziehen. Die Aussagekraft

[3]mit einem geringen Datenverlust, der bei der Gleitzahlumrechnung des segmentbasierten in den visierpunktbasierten Ansatz entsteht, der aber innerhalb eines tolerierbaren Bereichs liegt (Amann & Borrmann, 2015a).

solcher Metriken ist jedoch begrenzt und kann nicht als absolutes Maß betrachtet werden, sondern nur als ein Indikator.

2.6 Zusammenfassung

Building Information Modeling nimmt eine zunehmend wichtigere Rolle bei der Planung und Ausführung von Infrastrukturprojekten ein. Dabei stützen sich Bauwerksmodelle auf die Modellierungsansätze, die aus der traditionellen Trassierungsplanung mittels Höhe-, Lageplan und Querprofilen stammen. In diesem Kapitel wurden verschiedene Bauwerksmodelle aus dem Bereich des Infrastrukturbaus vorgestellt. Dabei wurde auf die fachlichen und technologischen Grundlagen diese Modelle eingegangen und verschiedene Datenaustauschformate für Trassierungsdaten beschrieben, die heute in der Praxis verwendet werden. Da die Anforderungen und Erwartungen an diese Bauwerksmodelle ständig zunehmen, um die Digitalisierung des Bauwesens weiter voranzutreiben, befinden sich die zugrundeliegenden Datenmodelle in einem ständigen Wandel. Mit diesem Wandel geht oft eine Komplexitätssteigerung einher, wie an den Datenmodellen IFC, OKSTRA und LandXML verdeutlicht wurde. Der in diesem Kapitel aufgezeigte instanzbasierte Ansatz zur Messung der Komplexität von Bauwerksdatenmodellen ermöglicht es, verschiedene Bauwerksdatenmodelle miteinander in Bezug auf ihre Komplexität zu vergleichen. Die Steigerung der Komplexität von Bauwerksmodellen stellt Standardisierungsgremien, Softwareentwickler und Anwender gleichermaßen vor immer größere Hürden bei der Definition, Umsetzung und Anwendbarkeit solcher Datenformate. Um auf Dauer mit diesem Komplexitätsanstieg umgehen zu können, müssen Ansätze entwickelt werden, welche die Handbarkeit einer zunehmenden Komplexitätssteigerung in diesen Datenmodellen erlauben. Im Rahmen dieser Arbeit wird ein Ansatz vorgestellt, der es erlaubt, den Datenaustausch auf einer abstrakteren Ebene mittels des Austausches von Programmen zu definieren, und dadurch der wachsenden Komplexität entgegenzuwirken.

Kapitel 3

Verwandte Arbeiten

In dieser Arbeit wird der Vorschlag gemacht, Informationen über Bauwerke in Form von objektorientierten Programmen auszutauschen. Im Folgenden sollen verwandte Arbeiten mit einer ähnlichen Zielsetzung kurz dargestellt werden.

3.1 Dynamische Querprofile in OKSTRA

Querprofile werden im OKSTRA-Standard (Version 2.018) durch die Klasse Querprofil beschrieben. Ein Querprofil wird einer Achse über einen Stationierungswert zugeordnet. Dabei steht das Querprofil senkrecht auf der Achse und definiert ein 2D-Koordinatensystem. Das Querprofil selbst wird dabei durch die Klasse Profilline beschrieben. Eine Profillinie setzt sich aus verschiedenen Punkten, die durch die Klasse QP_Punkt beschrieben werden, zusammen und kann eine Linie sowie einen Flächenumring definieren. Die Punkte des Querprofils werden durch einen Rechtswert und einen Hochwert beschrieben. Abbildung 3.1 zeigt einen Ausschnitt eines UML-Klassendiagramms, das die eben erwähnten Klassen enthält.

Die derzeitige Beschreibung von Querprofilen hat das Problem, dass diese keinen Design-to-Design-Transfer gewährleistet. Das heißt, wird ein Querprofil zwischen verschiedenen Anwendungen ausgetauscht, so ist es nicht möglich, dieses Querprofil auf Basis der übergebenen Daten an neue Ausgangsbedingungen anzupassen. Sollte es beispielsweise zu Änderungen der Achslage oder Gradiente kommen, so lassen sich die Querprofile auf Basis der übergebenen Daten nur unzureichend neu berechnen. Dieser Mangel wurde u. a. in der Studie (Kornbichler, 1999) festgestellt. Dabei wurde eine frühere Version von OKSTRA-Querprofilen untersucht, jedoch hat sich diese in Bezug auf Querprofile bis zu ihrer heutigen Ausprägung (OKSTRA 2.018) nicht wesentlich verändert.

Durch das Forschungs- und Entwicklungsprojekt “Dynamisches Querprofil”, das die Bundesanstalt für Straßenwesen (BASt) initiierte, wurde nach Lösungen gesucht, wie die OKSTRA-Querprofilbeschreibung verbessert werden kann, um einem verlustfreien Informationsfluss bei Design-to-Design-Transfers gerecht zu

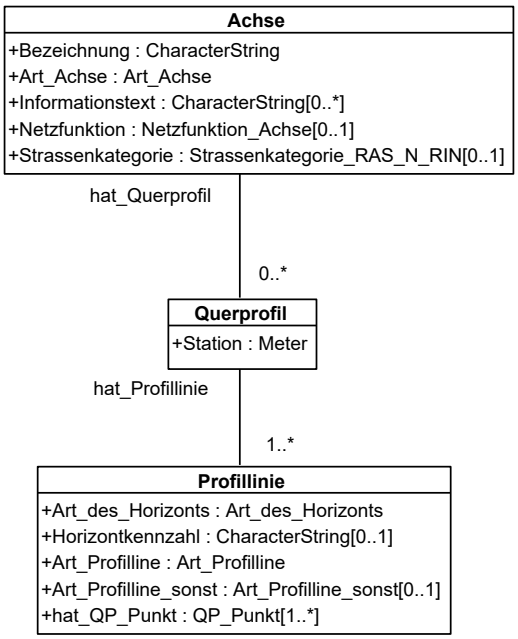


Abbildung 3.1: Querprofile werden im OKSTRA 2.018 durch die Klasse Querprofil beschrieben.

werden. Dieses Forschungsprojekt hat seinen Beginn im Jahr 1999 mit der Studie (Kornbichler, 1999) und wurde in dem Forschungs- und Entwicklungsauftrag "FE 09.122/2000/DGB OKSTRA Dynamisches Querprofil" in den Jahren 2000 bis 2004 fortgeführt (Feser & Rosenthal, 2004). Im Rahmen des OKSTRA-Symposiums in Berlin im Jahr 2007 wurde ein Prototyp des dynamischen Querprofils vorgestellt. Es wurde versucht, die prototypischen Entwicklungen durch einen Änderungsantrag in das OKSTRA-Datenmodell zu übernehmen, jedoch wurde dieser schlussendlich im Jahr 2013 abgelehnt, da der Zusatznutzen des dynamischen Querprofils inzwischen in Frage gestellt wurde (Thomsik, 2013).

Im Rahmen des Forschungsprojekts (FE 09.122/2000/DGB) wurde der Versuch unternommen, ein Modell für Querprofile zu entwickeln, das nicht die Ergebnisse der Modellierung, sondern die Konstruktionsschritte zur Bildung der Querprofile beschreibt. Dabei wurde eine objektorientierte Geometriesprache entwickelt, die zur Beschreibung von Querprofilen innerhalb des OKSRA-Standards genutzt werden kann. Die entwickelte Sprache trägt den Namen OKSTRA RQCode (Feser & Rosenthal, 2004) und ermöglich es, innerhalb des OKSTRA-Datenmodells Querschnitte geometrisch zu beschreiben. Die zugrundliegende Sprache stellt im Prinzip eine Weiterentwicklung dar, die statische (d. h. starre, fest vorgegebene) Regelquerschnitte (RQ) zu dynamischen Beschreibungen von Querschnitten erweitert, welche durch eine wohldefinierte Sprache spezifiziert sind.

Ziel von RQCode ist es nicht, die ausgewertete Geometriebeschreibung eines Querprofils beim Datenaustausch zu übertragen, sondern den Aufbau eines Querprofils mittels einer objektorientieren Sprache zu beschreiben. Dadurch ist es möglich, dass sich ein Querprofil an sich ändernde Randbedingungen automatisch anpasst.

RQCode setzt auf Visual Basic (Version 6) auf. Visual Basic wurde u. a. gewählt, weil dafür Übersetzer und Entwicklungswerkzeuge vorhanden sind, es objektorientiert ist (genau wie der OKSTRA-Standard selbst) und relativ einfach erlernbar ist. RQCode erbt alle Sprachfeatures von Visual Basic (VB), sieht jedoch nur die Verwendung einer Teilmenge des Sprachumfangs von VB vor. RQCode erlaubt die Verwendung einer Menge vordefinierter Objekte (RQLinie, RQBegrenzungslinie, RQPunkt, RQGeomOrt, RQVariable, RQInterface), mit deren Hilfe ein Querprofil beschrieben werden kann. Die Klasse RQLinie stellt beispielsweise Methoden bereit, die es ermöglichen, polygonale Linienzüge zu erstellen (im Sinne einer Profillinie). Der RQCode kann dabei Bezug auf Ausgangsdaten nehmen und abhängig von diesen Randbedingungen flexibel reagieren. Mehr Details dazu finden sich in (Kornbichler, 2000). Beispielsweise kann ein RQCode-Programm, abhängig von den Randbedingungen, flexibel darauf reagieren, ob ein Damm oder Einschnitt ausgebildet werden muss. Ein Problem, das sich bei diesem Ansatz ergibt, ist, dass Änderungen am Querprofil, zum Beispiel die Änderung der Böschungsform, nur möglich sind, indem der Quellcode des entsprechenden Querprofils modifiziert wird. Dies ist für typische Anwender kein praktikabler Ansatz und wurde im Rahmen des Forschungsprojektes als zu aufwändig und nicht durchsetzbar bewertet (Feser & Rosenthal, 2004). Basierend auf diesen Erkenntnissen wurde ein fachliches Baustein-Modell (FBM) entwickelt und der Ansatz, basierend auf RQCode, verworfen. Das FBM wurde schließlich aber ebenfalls nicht in den OKSTRA-Standard übernommen.

3.2 Die Datenmodellierungssprache EXPRESS

Der Sprachumfang der EXPRESS-Language ist relativ umfangreich. Dieser wird genutzt, um die Datenkonsistenz bzw. Datenintegrität zu wahren, den Wertebereich von Attributen einzuschränken und die Werte von abgeleiteten Attributen zu berechnen. Das Ziel, Informationen über Bauwerke auf Basis einer Programmiersprache auszutauschen, wird durch Derived-Attribute teilweise realisiert (vgl. Abschnitt 2.3.1), ist jedoch konzeptionell in seiner Umsetzung beschränkt. Da die EXPRESS-Sprache nicht mit diesem Designziel umgesetzt wurde, ist dies auch nicht weiter verwunderlich. Um beispielsweise den RQCode-Anwendungsfall der Querschnitte mittels der EXPRESS-Sprache beschreiben zu können, müsste erst ein geeignetes Konzept entwickelt werden, wie dieses in den EXPRESS-Kontext integriert werden kann. EXPRESS bietet hier keine geeignete Infrastruktur an, wie sie zum Austausch solcher Vorschriften erforderlich ist. Zudem kommt erschwerend hinzu, dass die Unterstützung der EXPRESS-Sprache durch Softwarewerkzeuge (wie z. B. Übersetzer, Modellierungssysteme oder Bibliotheken) sehr gering ist, was eine Weiterentwicklung in diesem Bereich stark erschwert.

3.3 Graphic Description Language

Die Software ArchiCAD, die von der Firma Graphisoft entwickelt wird, bietet eine eigene Skriptsprache mit dem Namen GDL (Graphic Description Language) an

(Graphisoft, 2018). Diese wird seit der ersten Version, die im Jahr 1984 erschienen ist, von ArchiCAD unterstützt (Nicholson-Cole, 2001). Dabei handelt es sich um eine Programmiersprache, die eine hohe Ähnlichkeit mit einem BASIC-Dialekt aufweist. Diese kann verwendet werden, um parametrische 3D-Objekte wie z. B. Türen, Fenster oder Treppen geometrisch zu beschreiben. Folgendes Listing zeigt ein Beispiel für ein GDL-Programm.

```
1  PRISM  4, 1,
2         3,  0,
3         3,  3,
4         −3, 3,
5         −3, 0
6  ADDZ −1
7  MUL 0.66667, 0.66667,
8  PRISM  4, 1,
9         3,  0,
10        3,  3,
11        −3, 3,
12        −3, 0
13 MUL 0.66667, 0.66667,
14 PRISM  4, 1,
15        3,  0,
16        3,  3,
17        −3, 3,
18        −3, 0
```

Abbildung 3.2 zeigt das Ergebnis dieses GDL-Programms. GDL-Programme werden auch als Scripte bezeichnet.

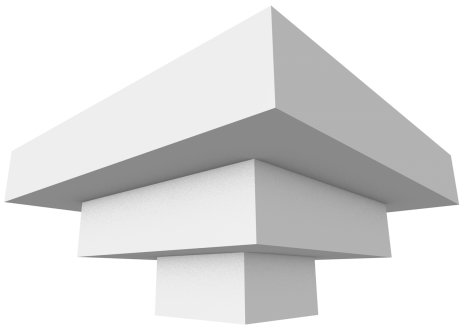


Abbildung 3.2: Ergebnis eines GDL-Programms

Neben 3D-Elementen können auch 2D-Elemente mittels GDL erzeugt werden. GDL kann auch benutzt werden, um beispielsweise Skripte für die Mengenermittlung zu formulieren. Die Syntax der GDL umfasst verschiedene Schleifenarten, bedingte Ausdrücke, Sprungbefehle (GOTO) und Variablendeklarationen. GDL-Skripte können mit Parameterwerten von außen besetzt werden, um so, abhängig von einer Benutzereingabe, verschiedene Objekte zu generieren.

GDL ist eine Sprache mit starkem Fokus auf die parametrische Modellierung, basierend auf Grundbausteinen, die die ArchiCAD-Umgebung bereitstellt. Der

Sprachkern ist sehr klein gehalten und die Sprache dadurch schnell lernbar. Durch die geometrischen Grundbausteine können in kurzer Zeit Ergebnisse erzielt werden. Jedoch ist GDL durch seinen Fokus und die entsprechende Integration in die ArchiCAD-Umgebung stark in den möglichen Anwendungsfällen limitiert und nicht produktübergreifend nutzbar.

3.4 Parametric IFC

Parametrisches Wissen kann auf Basis der IFC nur sehr eingeschränkt integriert und ausgetauscht werden. Abgesehen von Standardfällen für Fenster, Türen, CSG-Geometrien und (Stahl-)Profilen ist es nicht möglich, parametrisches Wissen in einem IFC-Modell zu transportieren. Da viele Softwaresysteme jedoch in der Lage sind, parametrische Daten zu verarbeiten, ist es von Vorteil, auch dieses parametrische Wissen beim Datenaustausch zu berücksichtigen (Ji *et al.*, 2013).

In einer Zusammenarbeit zwischen den Firmen AEC3 und RDF wurde IFC 2.3 erweitert, um auch parametrische Beschreibungskonzepte unterstützen zu können (buildingSMART, 2007b). An dieser Entwicklung war auch Jon Mirtschin von der Firma Geometry Gym beteiligt. Die Schemaerweiterung wurde auf den Webseiten von buildingSMART als Parametric IFC veröffentlicht (buildingSMART, 2007a). Im Kern dieser Erweiterung (Parametric IFC) stehen die zwei neuen Entitäten IfcParametricBinding und IfcParametricFormula.

Mit der Entität IfcParametricBinding kann ein Attribut einer anderen Entität, z. B. das Attribut Depth der Entität IfcExtrudedAreaSolid, referenziert werden. Eine Instanz der Entität IfcParametricBinding hat dabei selbst ein Attribut, mit dem ein beliebiger Name für dieses festgelegt werden kann. Der hierbei festgelegte Name dient als Variablenname, der in benutzerdefinierten parametrischen Formeln verwendet werden kann.

Eine IfcParametricFormula-Instanz beschreibt eine parametrische Formel. Diese besitzt einen Ausgabewert und eine beliebige Anzahl von Eingabewerten. Die konkrete Zuordnung von Eingabe- und Ausgabewerten dieser Formeln zu Werten von Entitäten erfolgt dabei durch IfcParametricBinding-Instanzen. Das Ergebnis der Auswertung der parametrischen Formel, die durch die IfcParametricFormula-Instanz definiert wird, wird als Wert für das Attribut gesetzt, auf welches das entsprechende Ausgabe-Binding (IfcParametricBinding) referenziert. Als parametrische Formeln sind dabei u. a. vom Benutzer definierte, einfache mathematische Formeln zulässig, z. B.:

deckLocationZ−foundLocationZ−overhandDeckAbut

Die Variablennamen (deckLocationZ, usw.) beziehen sich hier jeweils auf Instanzen der Entität (IfcParametricBinding), die wiederum auf Attribute bestimmter Entitäten referenzieren. Konstanten können ebenfalls innerhalb eines parametrischen Ausdrucks verwendet werden. Diese werden mit der Entität IfcParametricConstant beschrieben. Ein detailliertes Beispiel, basierend auf Parametric IFC, findet sich in (Kuloyants, 2014).

Parametric IFC erlaubt es, mathematische Regeln, basierend auf Variablennamen und mathematischen Operatoren, abzubilden. Außerdem ermöglicht das Konzept, mit geringem Aufwand einfache mathematische Abhängigkeiten zwischen verschiedenen Objekten zu beschreiben. Für ein parametrisches Modellierungssystem fehlen hier jedoch noch einige Erweiterungen (u. a. Constraints), die z. B. in (Ji *et al.*, 2013) beschrieben werden. Die Mächtigkeit dieses Ansatzes ist natürlich in seinen Ausdrucksmöglichkeiten beschränkt, jedoch ist die Realisierbarkeit des Ansatzes und die Erlernbarkeit für den Benutzer relativ überschaubar.

Der Ansatz wurde so gestaltet, dass dieser abwärtskompatibel mit dem existierenden IFC 2x3 Standard ist. Zudem kann der parametrische Anteil von einer Anwendung ignoriert werden, sofern keine geometrischen Änderungen am Modell vorgenommen werden sollen. In diesem Fall kann einfach mit den ausgewerteten Daten des Modells gearbeitet werden, die ebenfalls im Modell zusätzlich gespeichert sind.

3.5 Erweiterung der operationellen Spezifikation auf Basis von STEP nach Bühlmann

In der Dissertation (Bühlmann, 1996) werden STEP-Erweiterungen eingeführt mit dem Ziel, mehr semantische Informationen in den Datenaustausch einfließen zu lassen. Der Autor stellt fest, dass „Datenmodelliersprachen Zahlenwerte und nicht evaluierte Ausdrücke zu stark trennen“. Er schlägt daher vor, algebraische Ausdrücke und algorithmische Codeanweisungen als zusätzliche semantische Informationen in den Datenaustausch einfließen zu lassen. Die vorgestellten Vorschläge ermöglichen es u. a., Funktionen und Methoden innerhalb von Instanzdaten auszutauschen, die während der Modellierzeit nicht bekannt sind. Folgendes Beispiel, das aus dieser Arbeit übernommen wurde, soll die Funktionsweise von Funktionen auf Instanzebene veranschaulichen. Das Beispiel definiert zunächst eine Entität für Schraubenobjekte:

```
1  ENTITY Schraube;
2      GewindeL: REAL;
3      GewindeD: REAL;
4      KopfD: REAL;
5      SchraubenL: REAL;
6
7      WHERE
8          GewindeL > 0.0;
9          SchraubenL > GewindeL;
10         GewindeD < KopfD;
11         GewindeD > 0;
12 END_ENTITY;
```

Zusätzlich wird eine Entität für Flansch-Objekte eingeführt:

```
1  ENTITY Flansch;
2      -- number of screws to
3      -- mount flange
```

```
4       SchraubenN: INTEGER;
5       SchraubenD: REAL;
6
7       MittelD:
8           FUNCTION(REAL):REAL;
9       AussenD: REAL;
10      Hoehe: REAL;
11
12      WHERE
13          SchraubenN > 0;
14          SchraubenN MOD 4 = 0;
15          Hoehe > 0.1 * AussenD;
16  END_ENTITY;
```

Die Definition der Entität für Schrauben weicht noch nicht von einer Standard-EXPRESS-Syntax ab. Bei der Definition der Entität Flansch taucht jedoch beim Attribut MittelD eine Abweichung zur Standard EXPRESS-Syntax auf. Hierbei handelt es sich um die Funktionserweiterung, die in der Arbeit von Bühlmann vorgestellt wird. Diese erlaubt es, Attribute durch Funktionen berechnen zu lassen, die in einer STEP-P21-Datei definiert werden. In dem gezeigten Beispiel erwartet die Funktion einen REAL-Wert als Eingangsparameter und liefert einen REAL-Wert als Rückgabewert. Neben der Modifikation der EXPRESS-Grammatik wurde von Bühlmann auch die Grammatik von STEP-P21-Dateien entsprechend angepasst. Diese Modifikationen erlauben es, folgenden Datenaustausch auf Basis einer STEP-P21-Datei durchzuführen:

```
1   PARAMETER gl, sd, h: REAL;
2
3   DATA
4   #10= Flansch(12, sd, #40, 32.4, h);
5   #20= Schraube(gl, sd, #30(#10, sd),#20.GewindeL*1.2);
6   #30= FUNCTION(f: Flansch; s: Schraube): REAL;
7           RETURN MIN(s.KopfD, 0.8*(f.AussenD-f.MittelD)/2);
8         END_FUNCTION;
9   #40= FUNCTION(arg:REAL): REAL;
10          RETURN 0.6*arg;
11        END_FUNCTION;
12  END_DATA
```

In Zeile 1 der vorhergehenden STEP-P21-Datei werden Parameter definiert. Dabei handelt es sich um eine Erweiterung, die es erlaubt, freie Variablen mit einem entsprechenden Namen und Typen zu spezifizieren. Dabei handelt es sich um Variablen, die erst zu einem späteren Zeitpunkt festgelegt werden. Bühlmann führt diese Erweiterung im Kontext von generischen Objekten für Teilebibliotheken ein. Dabei stehen Bauteile im Fokus, die sich nur geringfügig durch bestimmte Parameter unterscheiden. Tatsächlich wird bei obigem Beispiel keine konkrete Instanz ausgetauscht, sondern nur eine Schablone, auf deren Basis mittels gegebener Parameter (gl, sd und h) entsprechende konkrete Bauteile herstellerneutral ausgetauscht werden können. Die Parametererweiterung ermöglicht es, Abhängigkeiten zwischen verschiedenen Entitäten zu berücksichtigen. Beispielsweise müssen der

Schraubendurchmesser des Flansches und der Durchmesser der Schraube zueinander passen. Dies wird durch den Parameter (freie Variable) sd abgedeckt.

Die Flansch-Instanz in Zeile 4 spezifiziert als drittes Attribut eine Funktion (siehe EXPRESS-Definition des Attributs MittelD der Entität Flansch). Das Attribut verweist auf die Funktion, die in Zeile 9 definiert ist (#40). Der Übergabeparameter, der dieser Funktion übergeben wird, muss von außen bereitgestellt werden, d. h. von der Applikation, die dieses Attribut auswerten möchte. In Zeile 5 (Schrauben-Instanz) wird als Angabe für das dritte Attribut (KopfD) auf die Funktion, die in Zeile 6 definiert ist (#30), mit entsprechenden Funktionsparametern (#10, sd) verwiesen. Dieses Attribut (KopfD) ist keine Referenz auf eine Funktion, sondern einfach nur ein REAL-Attribut. Der Wert dieses Attributes berechnet sich durch Aufruf der Funktion, die in Zeile 6 definiert ist, mit entsprechenden Funktionsparametern. Der Ansatz von Bühlmann unterstützt außerdem einfache algebraische Ausdrücke, wie in Zeile 5 dargestellt ist (#20.GewindeL∗1.2).

Funktionen werden durch einen Interpreter ausgewertet. Dieser Interpreter muss in jedes Programm, das Daten mit der beschriebenen Erweiterung lesen will, miteingebunden werden.

Für die Beschreibung von parametrischen Bauteilen ist diese Funktionserweiterung sinnvoll. Neben dieser Erweiterung durch funktionswertige Attribute beschreibt Bühlmann auch eine STEP-Erweiterung durch Methoden.

Er schreibt dazu in (Bühlmann, 1996):

> „... eine vollständige Definition von Datentypen [besteht] aus zwei Teilen, einer strukturellen und einer operationellen Spezifikation. [...] [H]eutige Datenmodelliermittel [beschränken] sich auf die strukturelle Beschreibung [...]. Als Konsequenz fehlen die semantisch bedeutungsvollen, operationellen Teile. Sie sind einzig in den Anwendungen programmiert und führen so bei jeder Schemaänderung zu Problemen. Daneben zeigt es sich, dass nicht nur die Dokumentation des Datenschemas unvollständig ist, sondern auch Mehrfachimplementationen derselben Funktionalität in verschiedenen Applikationen vorkommen."

Auch hier soll das entsprechende Beispiel aus (Bühlmann, 1996) aufgegriffen werden. Im Rahmen dieses Beispiels soll das Volumen eines geometrischen Objektes wie beispielsweise eines Flansches berechnet werden:

```
1  FUNCTION BBVolGeom(arg: Geometry): REAL;
2      RETURN 0.0; −− general object;
3  END_FUNCTION;
4
5  ENTITY Geometry
6      ABSTRACT SUPERTYPE OF ONEOF(Flansch, Schraube);
7  DERIVE
8      BoundBoxVol: FUNCTION(Geometry): REAL =
```

```
9           BBVolGeom;
10 END_ENTITY;
```

Die Entät Flansch erbt von der abstrakten Entität Geometry und überschreibt die Methode BBVolGeom entsprechend mit einer eigenen Implementierung:

```
1  FUNCTION BBVolFlansch(arg: Flansch): REAL;
2      RETURN arg.AussenD*arg.AussenD*arg.Hoehe;
3  END_FUNCTION;
4
5  ENTITY Flansch;
6       [...]
7      DERIVE
8      SELF\Geometry.BoundBoxYol:FUNCTION(Flansch):REAL
9           = BBYolFlansch;
10      [...]
11 END_ENTITY;
```

Die Verwendung (Aufruf) einer Methode wird im folgenden Beispiel gezeigt:

```
1  x: Flansch;
2  volume: REAL;
3
4  x := Flansch (...);
5  volume := x.BoundBoxVol(x); -- method call
```

Die Arbeit von Bühlmann bietet sehr interessante Ansätze, um Informationen über Bauwerke in Form von Programmen (Funktionen und Methoden) auszutauschen.

Im Bereich der Beschreibung von parametrischen Bauteilen wird mittels funktionaler Attribute und freien Variablen die Möglichkeit geboten, beliebige Parameter für Bauteile einzuführen. Eine Applikation, die einen solchen Datenimport bewerkstelligen muss, weiß jedoch aufgrund der fehlenden Schnittstellendefinition nicht, welche Parameter erwartet werden. Dies führt dazu, dass eine Applikation mit diesen Parametern nur in eingeschränkter Weise umgehen kann, da diese keine Vorkenntnisse über das zugrundeliegende Modellobjekt besitzt. Dies mag für parametrische Bauteilbeschreibungen, die nur wenige und triviale Datentypen besitzen, ausreichend sein, kann aber bei komplexeren parametrischen Bauteilen nur schwer von einer Anwendung ausgewertet werden, da der Fluss der Datenkommunikation zwischen einer Anwendung und den parametrischen Bauteilbeschreibungen nicht detailliert spezifiziert ist.

Der Vorschlag, das Datenmodell durch Methoden zu erweitern, sieht keine Erweiterung von Entitäten durch neue Attribute außerhalb der Modellierzeit (Definitionsphase des Schemas) vor. Außerdem können auch keine neu definierten Entitäten/Klassen im Rahmen der Realisierung von Methoden bzw. Funktionen eingeführt und genutzt werden. Es können nur Attribute und Entitäten/Klassen verwendet werden, die während der Modellierzeit berücksichtig worden sind. Bei komplexeren Programmen stellt dieser Umstand ein Hemmnis dar, da hier eine

Erweiterung des operationellen Teils oft mit einer Erweiterung bzw. Anpassung des strukturellen Teils einhergeht.

3.6 Weitere Arbeiten

Im Kontext von BIM im Infrastrukturbau und dem Austausch von semantischen Informationen auf Basis von Programmen in Quelltextform gibt es noch einige weitere Arbeiten.

Beispielsweise wird im Rahmen des OpenBrIM-Projekts die Sprache ParamML verwendet, um parametrische Brückenbauteile zu modellieren (Jeong *et al.*, 2017). Ein Beispiel eines Brückenwiderlagers, das auf Basis von OpenBrIM und ParamML modelliert wurde, ist in Abbildung 3.3 zu sehen.

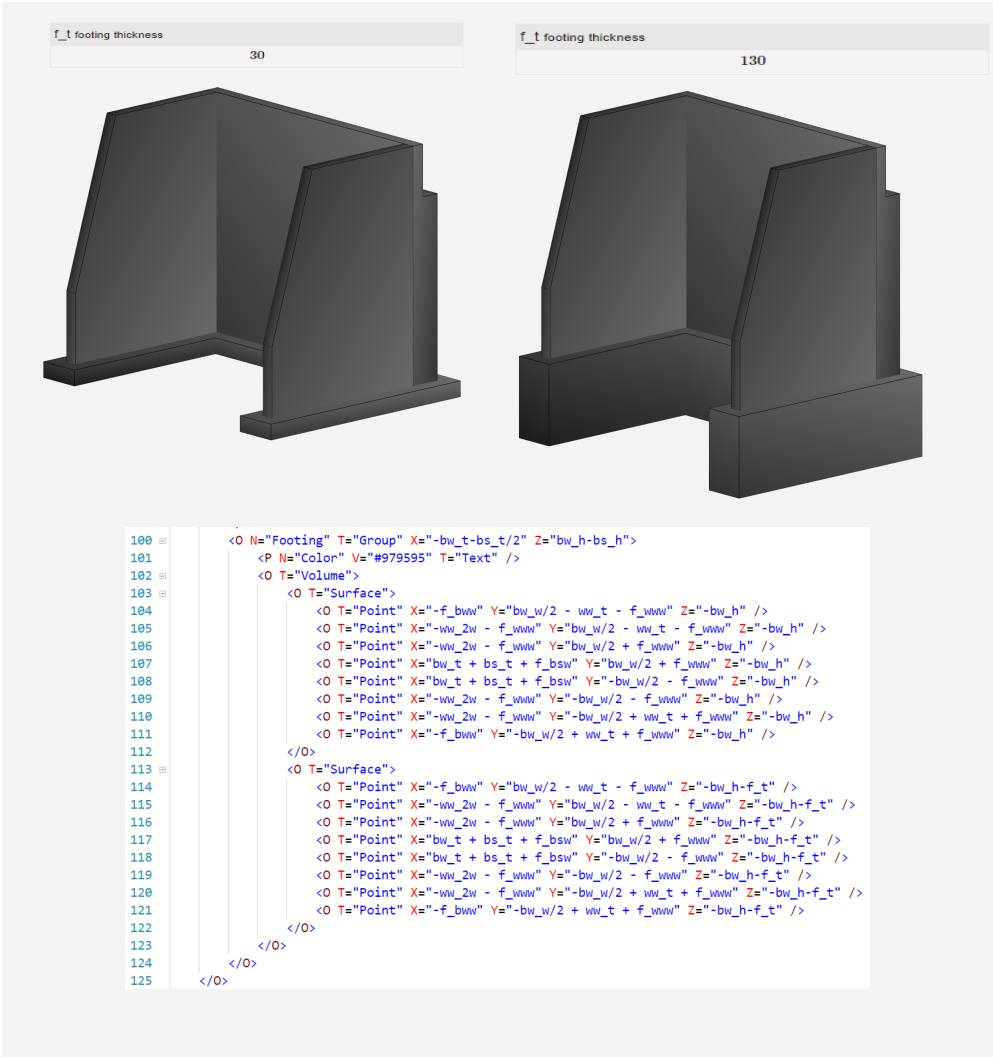
Neben Sprachen für die Beschreibung von parametrischen Objekten gibt es auch einige Abfrage-Sprachen (Query Languages), die Suchanfragen auf Gebäudeinformationsmodelle bieten wie z. B. die Sprache BIMQL (Mazairac & Beetz, 2013) oder QL4BIM (Daum & Borrmann, 2015; Daum *et al.*, 2016). Diese spielen jedoch im Rahmen dieser Arbeit eine untergeordnete Rolle.

Die verschiedenen Strategien, um Programme in Produktdatenmodelle zu integrieren, reichen von sehr allgemeinen Ansätzen (vgl. (Bühlmann, 1996)) bis hin zu sehr spezifischen Lösungen für spezielle Zwecke wie z. B. die Beschreibung von parametrischen Bauteilen. Die technologischen Grundlagen der Sprachen stützen sich dabei auf STEP, XML, existierende Programmiersprachen oder auch auf herstellerspezifische Technologien (siehe GDL).

3.7 Zusammenfassung

Ein wesentlicher Aspekt dieser Arbeit besteht darin, herstellerneutrale Bauwerksmodelle durch Programme zu erweitern, um dadurch flexibel auf Änderungen und Erweiterungen beim Datenaustausch reagieren zu können, ohne lange Standardisierungsprozesse oder Softwareanpassungen durchführen zu müssen. Die vorgestellten verwandten Arbeiten können diese Anforderungen nur teilweise erfüllen.

Das dynamische Querprofil in OKSTRA auf Basis von RQCode und vordefinierten Objekten zur Beschreibung von Querschnittprofilen wird den genannten Anforderungen zwar in Bezug auf Querprofilbeschreibungen gerecht, jedoch ist dieser Ansatz stark auf diesen Anwendungsfall beschränkt. Beispielswiese sind die vordefinierten Objekte stark an die Randbedingungen von Querprofilen angepasst und die Nutzung der vordefinierten Klassen für andere Verwendungszwecke nicht vorgesehen bzw. möglich. Zudem fehlt ein generelles Konzept, das es erlauben würde, RQCode auch für andere Anwendungsszenarien zu verwenden. Der in dieser Arbeit vorgestellte Ansatz adressiert diese Lücken und zeigt einen flexiblen Ansatz auf, der nicht auf eine bestimmte Funktionalität oder ein bestimmtes Anwendungsszenario beschränkt ist.




```
100      <O N="Footing" T="Group" X="-bw_t-bs_t/2" Z="bw_h-bs_h">
101          <P N="Color" V="#979595" T="Text" />
102          <O T="Volume">
103              <O T="Surface">
104                  <O T="Point" X="-f_bww" Y="bw_w/2 - ww_t - f_www" Z="-bw_h" />
105                  <O T="Point" X="-ww_2w - f_www" Y="bw_w/2 - ww_t - f_www" Z="-bw_h" />
106                  <O T="Point" X="-ww_2w - f_www" Y="bw_w/2 + f_www" Z="-bw_h" />
107                  <O T="Point" X="bw_t + bs_t + f_bsw" Y="bw_w/2 + f_www" Z="-bw_h" />
108                  <O T="Point" X="bw_t + bs_t + f_bsw" Y="-bw_w/2 - f_www" Z="-bw_h" />
109                  <O T="Point" X="-ww_2w - f_www" Y="-bw_w/2 - f_www" Z="-bw_h" />
110                  <O T="Point" X="-ww_2w - f_www" Y="-bw_w/2 + ww_t + f_www" Z="-bw_h" />
111                  <O T="Point" X="-f_bww" Y="-bw_w/2 + ww_t + f_www" Z="-bw_h" />
112              </O>
113              <O T="Surface">
114                  <O T="Point" X="-f_bww" Y="bw_w/2 - ww_t - f_www" Z="-bw_h-f_t" />
115                  <O T="Point" X="-ww_2w - f_www" Y="bw_w/2 - ww_t - f_www" Z="-bw_h-f_t" />
116                  <O T="Point" X="-ww_2w - f_www" Y="bw_w/2 + f_www" Z="-bw_h-f_t" />
117                  <O T="Point" X="bw_t + bs_t + f_bsw" Y="bw_w/2 + f_www" Z="-bw_h-f_t" />
118                  <O T="Point" X="bw_t + bs_t + f_bsw" Y="-bw_w/2 - f_www" Z="-bw_h-f_t" />
119                  <O T="Point" X="-ww_2w - f_www" Y="-bw_w/2 - f_www" Z="-bw_h-f_t" />
120                  <O T="Point" X="-ww_2w - f_www" Y="-bw_w/2 + ww_t + f_www" Z="-bw_h-f_t" />
121                  <O T="Point" X="-f_bww" Y="-bw_w/2 + ww_t + f_www" Z="-bw_h-f_t" />
122              </O>
123          </O>
124      </O>
125  </O>
```

Abbildung 3.3: Beispiel eines Brückenwiderlagers, das auf Basis von ParamML modelliert wurde. Der Wert der Variablen `f_t` (footing thickness) wurde von 30 auf den Wert 130 geändert und führt aufgrund der Auswertung des ParamML Dokuments zu einer modifizierten Geometrie.

EXPRESS verfügt als Datenmodellierungssprache über Möglichkeiten, Programme zu formulieren, die die Datenintegrität gewährleisten sollen, und zudem aus vorhandenen Attributen neue abgeleitete Attribute berechnen können. Jedoch ist auch dieser Mechanismus konzeptionell in seiner Umsetzung beschränkt, da EXPRESS auf Schemaebene angesiedelt ist und damit keinen Programmaustausch auf Instanzebene zur Laufzeit ermöglicht. Der in dieser Arbeit beschriebene Ansatz arbeitet auf Instanzebene und löst dieses Problem.

Die Graphic Description Language (GDL) ist durch ihren Fokus und die entsprechende Integration in die Archicad-Umgebung stark in den möglichen Anwendungsfällen limitiert. Durch die starke Kopplung an ein konkretes Softwareprodukt ist diese auch nur bedingt als herstellerübergreifende Lösung für einen programmbasierten Datenaustausch verwendbar. Im Vergleich dazu wird in dieser Arbeit ein herstellerneutraler Ansatz vorgeschlagen.

Parametric IFC hat einen starken Fokus auf die Beschreibung von parametrisierten Bauteilen. Abhängigkeiten zwischen Bauteilten lassen sich hierbei durch einfache Formeln beschreiben, die Attribute unterschiedlicher Bauteileelemente referenzieren können. Der Ansatz ist auf die Beschreibung von paramterischen Abhängigkeiten beschränkt und die Ausdrucksmöglichkeiten sind sehr limitiert. Komplexere Abhängigkeiten oder ein Variantenmanagement, das unterschiedliche Ausprägungen eines parametrisierten Bauteils vorsieht, sind damit nur schwer zu realisieren.

In (Bühlmann, 1996) wird ein Ansatz vorgestellt, der dem in dieser Arbeit verfolgten Ansatz am nächsten kommt. Er besitzt jedoch einige Lücken, die durch den in dieser Arbeit vorgestellten Ansatz geschlossen werden. Unter anderem fehlt im Ansatz von Bühlmann eine klare Schnittstellenbeschreibung für ausgetauschte Programme, die für eine Verarbeitung durch eine empfangende Softwareapplikation zwingend nötig ist, insofern die Funktionsweise eines Programmes nicht impliziert bekannt ist. Darüber hinaus sieht der Ansatz von Bühlmann keine Erweiterung von Entitäten durch neue Attribute außerhalb der Modellierzeit vor. Außerdem können auch keine neu definierten Entitäten bzw. Klassen im Rahmen der Realisierung von Methoden bzw. Funktionen eingeführt und genutzt werden, was bei der Formulierung umfangreicherer Programme ein Hemmnis darstellt.

Kapitel 4

Konzeptioneller Ansatz

Ein Ziel, das durch den Einsatz von Bauwerksdatenmodellen (eines Building Information Models) angestrebt wird, ist der effiziente Informationsaustausch zwischen den beteiligten Akteuren im gesamten Lebenszyklus eines Bauwerks (Eastman *et al.*, 2008; Borrmann *et al.*, 2015). Derzeit etablierte Datenmodelle weisen dabei einige Unzulänglichkeiten auf, die im folgenden Abschnitt anhand eines Beispiels erläutert werden sollen. Zunächst wird beschrieben, wie der Austausch von Bauwerksdatenmodellen heute realisiert wird. Dann wird auf die Einschränkungen dieses Datenaustausches eingegangen. Zum Schluss wird ein Konzept vorgestellt, das Verbesserungen für den Austausch von Bauwerksdatenmodellen vorschlägt.

4.1 Austausch von Bauwerksdatenmodellen

Gegeben sei das Austauschszenario aus Abbildung 4.1. Dabei sollen zwischen einer Applikation A (AutoCAD Civil 3D (Autodesk, 2018)) und einer Applikation B (TUM Open Infra Platform (CMS, 2018b)) Trassierungsdaten eines Verkehrswegs und ein digitales Geländemodell (DGM) ausgetauscht werden. Dazu stehen verschiedene Möglichkeiten bereit. Für den Austausch von Planungsdaten einer Trassierung bietet sich eine Reihe von Datenaustauschformaten an, z. B. LandXML, IFC4X1 oder OKSTRA. Auf internationaler Ebene hat sich hierbei das Datenformat LandXML etabliert. Auf nationaler Ebene innerhalb Deutschlands findet man im Bereich des Straßenbaus auch öfter das OKSTRA-Dateiformat vor. Mit IFC 4.1 steht seit Kurzem ein weiteres Datenformat bereit, das diesen Datenaustausch ebenfalls bewerkstelligen kann. Im Bereich der GIS-Systeme könnte hier auch noch das LandInfra Model (Implementierung des Metamodells InfraGML) Einsatz finden.

Die typischen Informationen, die zwischen den Systemen A und B ausgetauscht werden sollen, sind Trassierungs- und Geländemodelldaten. Die Trassierung basiert hier üblicherweise auf einem Lage- und Höhenplan. Der Lageplan setzt sich wiederum aus verschiedenen Trassierungselementen wie Geraden, Kreisbögen und Klothoiden zusammen. Für den Höhenplan werden, abhängig von seiner Reprä-

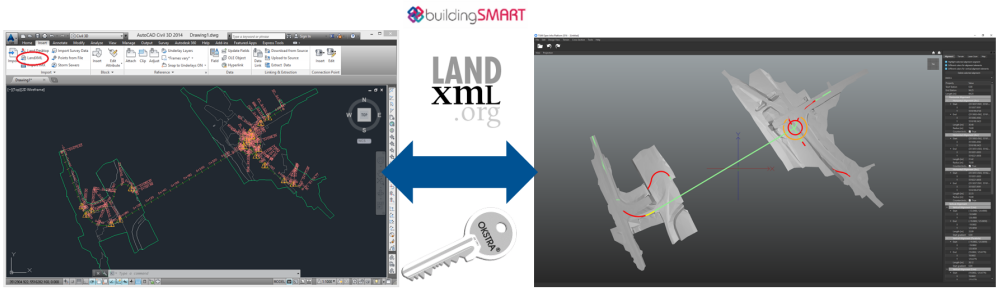


Abbildung 4.1: Für den Austausch von Trassierungsdaten steht eine Reihe von Datenaustauschformaten wie z. B. LandXML, IFC4X1 oder OKSTRA bereit.

sentation, entweder die Daten zu den einzelnen Segmenten, aus denen sich dieser zusammensetzt, oder entsprechend die Stationierungspunkte zusammen mit Radien der entsprechenden Ausrundungsparameter übertragen. Für ein digitales Geländemodell werden häufig Dreiecksnetze verwendet. Dabei werden beim Datenaustausch im Wesentlichen die Punkte des Dreiecksnetzes und dessen Vermaschung übertragen.

Ein Ausschnitt eines solchen Datenaustausches zeigt folgendes Listing:

```
1 <XYCurve direction="1.75917" x="50.0104" y="-83.2074">
2     <ClothoArc endCurvature="0" length="8.009" startCurvature="-0.0309793"/>
3     <ClothoArc endCurvature="0" length="3.51943" startCurvature="0.0143293"/>
4     <CircleArc curvature="0.00105004" length="128.811"/>
5     <Segment length="2.7"/>
6 </XYCurve>
```

Dabei handelt es sich um RoadXML-Daten. RoadXML ist ein Datenformat, das im Bereich von Fahrsimulatoren eingesetzt wird. Es ist selbst auf Basis eines XML-Schemas definiert und gibt dadurch die Struktur für Instanzdaten vor (Ducloux & Millet, 2009). Weiter definiert das XML-Schema, welche Datentypen in einer Instanzdatei vorkommen dürfen. An dieser Stelle wurde ein RoadXML-Beispiel aufgegriffen, weil es eine sehr kompakte Darstellungsform besitzt und im Wesentlichen widerspiegelt, wie die genannten Dateiformate (LandXML, IFC 4.1, OKSTRA) Lagepläne übertragen. Eine XYCurve gibt vor, dass es sich hierbei um einen Lageplan handelt. Die x- und y-Koordinaten des Startpunktes sowie die Startrichtung (direction) sind als Attribute vorgegeben. Das erste und zweite Segment beschreibt jeweils eine Klothoide (ClothoArc), das dritte einen Kreisbogen (CircleArc) und das letzte ein Geradenstück (Segment). Die verschiedenen Elemente des Lageplans enthalten relevante Daten wie z. B. die Start- und Endkrümmung (startCurvature, endCurvature) oder die Segmentlänge (length).

Durch Analyse des Datenaustausches kommt man zu dem Schluss, dass der Datenaustausch auf Instanzebene auf folgenden Elementen basiert:

- Einfache/wohldefinierte Datentypen (z. B. String, Integer, Real)
- Komplexe Datentypen (z. B. Listen, Arrays, Enumerationen)

- Benutzerdefinierte Datentypen (z. B. Klassen, Typ-Aliase)

Im obigen Beispiel ist XYCurve ein benutzerdefinierter Datentyp, der einen Startpunkt und eine Startrichtung sowie eine Liste (komplexer Datentyp) von Lageplansegmenten besitzt. Die Startrichtung ist durch einen einfachen Datentyp (Real) spezifiziert.

Der Datenaustausch auf Basis von IFC 4.1 läuft in ähnlicher Weise ab. Dort wird im Regelfall eine STEP-P21-Datei ausgetauscht. Folgendes Listing zeigt, wie in IFC 4.1 ein Kreissegment eines Lageplans ausgetauscht wird. Dazu wird die Entität IfcCircularArcSegment2D genutzt. Abbildung 4.2 zeigt zudem die Dokumentation der entsprechenden Entität IfcCircularArcSegment2D.

```
1 #43=IFCCIRCULARARCSEGMENT2D(
2     #44,
3     5.2461930831247052254,
4     30.484556939386848740,
5     15.000040369074330471,
6     .T.);
7 #44=IFCCARTESIANPOINT((0.,0.));
```

#	Attribute	Type	Cardinality	Description	G
IfcRepresentationItem					
	LayerAssignment	IfcPresentationLayerAssignment @AssignedItems	S[0:1]	Assignment of the representation item to a single or multiple layer(s). The *LayerAssignments* can override a *LayerAssignments* of the IfcRepresentation it is used within the list of *Items*. IFC2x3 CHANGE The inverse attribute *LayerAssignments* has been added. IFC4 CHANGE The inverse attribute *LayerAssignment* has been restricted to max 1. Upward compatibility for file based exchange is guaranteed.	X
	StyledByItem	IfcStyledItem @Item	S[0:1]	Reference to the IfcStyledItem that provides presentation information to the representation, e.g. a curve style, including colour and thickness to a geometric curve. IFC2x3 CHANGE The inverse attribute *StyledByItem* has been added.	X
IfcGeometricRepresentationItem					
IfcCurve					
	Dim :=IfcCurveDim(SELF)	IfcDimensionCount		The space dimensionality of this abstract class, defined differently for all subtypes, i.e. for IfcLine, IfcConic and IfcBoundedCurve.	X
IfcBoundedCurve					
IfcCurveSegment2D					
1	StartPoint	IfcCartesianPoint		The start point of the 2D curve as x/y coordinates defined by a 2D Cartesian point.	X
2	StartDirection	IfcPlaneAngleMeasure		The direction of the tangent at the start point. Direction value 0. indicates a curve with a start tangent along the positive x-axis. Values increases counter-clockwise, and decreases clockwise. Depending on the plane angle unit, either degree or radians, the sensible range is -360° ≤ n ≤ 360° (or -2π ≤ n ≤ 2π). Values larger then a full circle (>|360°| or >|2 π| shall not be used.	X
3	SegmentLength	IfcPositiveLengthMeasure		The length along the curve	X
IfcCircularArcSegment2D					
4	Radius	IfcPositiveLengthMeasure		The radius of the circular arc	X
5	IsCCW	IfcBoolean		(counter-clockwise or clockwise) as the orientation of the circular arc with Boolean="true" being counter-clockwise, or "to the left", and Boolean="false" being clockwise, or "to the right".	X

Abbildung 4.2: Ausschnitt eines Screenshots aus der IFC 4.1 Dokumentation (buildingSMART, 2018c), der die Attributliste des Elements IfcCircularArcSegment2D zeigt

Der erste Wert in der Zeile 2 (#44) referenziert auf den Startpunkt des Kreissegments. Dann folgen die Startrichtung, die Segmentlänge, der Radius und die Information, ob der Kreisbogen im oder gegen den Uhrzeigersinn orientiert ist. Das Format der STEP-P21-Datei basiert auf dem IFC 4.1 EXPRESS-Schema. STEP-P21-Dateien werden genutzt, um Instanzdateien auszutauschen. Der Datenaustausch basiert ebenfalls auf einfachen/wohldefinierten, komplexen und benutzerdefinierten Datentypen. Zudem unterstützt EXPRESS Derived-Attributes (hergeleitete Attribute), d.h. Attribute, die, basierend auf Daten des zugrundeliegenden Schemas, berechnet werden können. Dabei wird für die Herleitung eines Derived-Attributs auf der Schemaebene eine formale Regel definiert. Die Regel ist mithilfe der EXPRESS-Sprache definiert und Teil des EXPRESS-Schemas.

Die Entität IfcSIUnit besitzt beispielsweise ein solches Derived-Attribute. Der Name des Attributs lautet Dimensions und besitzt den Typ IfcDimensionalExponents, der anhand einer im EXPRESS-Schema definierten EXPRESS-Funktion IfcDimensionsForSiUnit hergeleitet werden kann. Diese EXPRESS-Funktion befindet sich im IFC 4.1-EXPRESS-Schema und ist im Folgenden verkürzt dargestellt:

```
1  FUNCTION IfcDimensionsForSiUnit
2  (n : IfcSIUnitName )     : IfcDimensionalExponents;
3    CASE n OF
4      METRE              : RETURN (IfcDimensionalExponents
5                                 (1, 0, 0, 0, 0, 0, 0));
6      SQUARE_METRE : RETURN (IfcDimensionalExponents
7                                 (2, 0, 0, 0, 0, 0, 0));
8      CUBIC_METRE    : RETURN (IfcDimensionalExponents
9                                 (3, 0, 0, 0, 0, 0, 0));
10     GRAM               : RETURN (IfcDimensionalExponents
11                                (0, 1, 0, 0, 0, 0, 0));
12      [...]
13   END_CASE;
14 END_FUNCTION;
```

Diese Funktion ist, wie bereits beschrieben, nicht Teil der STEP-P21-Instanz-Datei, sondern Teil des EXPRESS-Schemas. Auf Instanzebene werden keine Funktionen ausgetauscht.

Auch bei der Analyse des Datenaustausches von OKSTRA, LandXML und LandInfra kommt man zu der Erkenntnis, dass auf Instanzebene nur die Werte einfacher, komplexer und benutzerdefinierter Datentypen ausgetauscht werden.

4.2 Probleme beim Austausch von Bauwerksdatenmodellen

Der Datenaustausch von Bauwerksinformationsmodellen im Kontext von Infrastrukturprojekten stellt gleichermaßen Standardisierungskomitees, Anwender und Softwareentwickler vor große Herausforderungen. Damit überhaupt ein Datenaustausch erfolgen kann, ist das Zusammenwirken von sehr vielen Individuen

erforderlich. Der typische Ablauf bei der Umsetzung eines herstellerneutralen Datenaustausches läuft in folgenden Schritten ab:

- *Zunächst muss ein Standard definiert werden*: Verschiedene Softwarehersteller müssen sich auf ein einheitliches Austauschformat einigen. Im Rahmen von IFC gibt es dafür die Organisation buildingSMART International (bSI), die Erweiterungsvorschläge bewilligt und diese bei der Umsetzung organisatorisch unterstützt. BuildingSMART ist in sogenannte Chapters aufgeteilt. So gibt es beispielsweise ein deutsches Chapter (buildingSMART Deutschland), das sich unter dem Dach von buildingSMART International befindet. Weitere Chapters sind u. a. Australien, Kanada oder Korea. Eine Übersicht aller Chapters befindet sich unter (buildingSMART, 2018a). Dabei kann für jedes Land ein eigenes Chapter gegründet werden, wenn sich dort entsprechende Mitglieder finden. Mitglieder von bSI müssen nicht zwangsläufig in Chapters organisiert sein, sondern können auch unabhängig von einem Chapter Mitglied bei bSI werden. Im Regelfall werden im Rahmen von Erweiterungen des IFC-Standards sogenannte Expert Panels durchgeführt, in denen Mitglieder eines Chapters bzw. sonstige Mitglieder teilnehmen können. Dabei wird schrittweise in mehreren durchgeführten Expert Panels ein internationaler Konsens entwickelt, der letztlich in einer Erweiterung des IFC-Datenmodells mündet. Teilweise werden dabei auch erst auf nationaler Ebene Expert Panels durchgeführt, um so die Anforderungen und Vorstellungen eines Chapters zu erfassen. Die Ergebnisse werden dann im Rahmen eines internationalen Expert Panels aufgegriffen und international abgestimmt. Dies führt zu einem erheblichen Kommunikations- und Abstimmungsaufwand, der aber nötig ist, um mit allen Beteiligten zusammen einen gemeinsamen Standard zu entwickeln. Das IFC Alignment Projekt 1.0 (buildingSMART, 2014; buildingSmart Model-Support Group, 2015), das zum Ziel hatte , IFC 4.0 mit trassierungsspezifischen Informationen zu erweitern, hatte eine Projektlaufzeit von ca. einem Jahr. Darauf aufbauend wurde das IFC Alignment Projekt 1.1 gestartet, das u. a. Unterstützung für lineare Referenzierung, weitere Übergangskurven und bessere Geometrieunterstüzung für Infrastrukturbauwerke (u. a. IfcSectionedSolid. IfcSectionedSolidHorizontal) zum Ziel hatte und ebenfalls eine Projektlaufzeit von ca. einem Jahr in Anspruch genommen hat. Bei diesen beiden Projekten handelte es sich noch um relativ überschaubare Ergänzungen des IFC-Standards. Solche Standardisierungsvorhaben sind jedoch, wie beschrieben, sehr aufwändig, da viele Experten ein Mitspracherecht besitzen, was dazu führt, dass Standards von buildingSMART eine hohe internationale Akzeptanz besitzen, dies jedoch mit einem hohen zeitlichen Aufwand verbunden ist. Kurzum: Die Definition und Einigung auf einen Standard ist ein zeit- und kostenintensives Verfahren.

- *Softwarehersteller müssen den Standard implementieren*: Wenn ein Standard verabschiedet wurde, muss dieser erst durch Softwareprodukte unterstützt werden. Der beste Standard hilft ohne Softwareunterstützung nichts.

Softwareentwickler haben natürlich nur ein begrenztes Zeit- und Kostenbudget und können häufig nur einen Bruchteil der gewünschten Funktionalität ihrer Anwender realisieren. Bei einem neuen Standard ergibt sich dabei oft ein Henne-Ei-Problem: Softwareentwickler wollen keine Standards unterstützen, die vom Benutzer nicht verwendet werden. Benutzer können keine Standards verwenden, für die keine Software bereitsteht. Dies führt oft zu Verzögerungen bei der Umsetzung von Standards. Beispielsweise unterstützten lange Zeit viele Werkzeuge nur IFC 2.3 (buildingSMART, 2018b), obwohl schon IFC 4.0 zur Verfügung stand. In jedem Fall benötigt die Entwicklung von entsprechenden Import- und Exportfunktionalitäten einen gewissen Aufwand, der von den Herstellern von Softwareprodukten zu leisten ist.

– *Der Standard muss richtig verstanden und korrekt implementiert werden*: Auch wenn ein Standard definiert wurde und Softwareentwickler bereit sind, diesen umzusetzen, ist nicht sichergestellt, dass Softwareentwickler diesen Standard korrekt implementieren. Dies führt nicht zuletzt auf Anwenderseite zu Frustration, wenn der Datenaustausch nicht reibungsfrei funktioniert. Fehler können hier z. B. entstehen, weil die Dokumentation nicht eindeutig ist, da diese meist nur informal in textueller Beschreibung vorliegt. Dies kann Interpretationsspielräume lassen oder zu Missverständnissen führen. Abgesehen davon können auch Fehler bei der Implementierung (z. B. der Programmierung) gemacht werden.

– *Jemand muss mit der zunehmenden Komplexität des Modells umgehen können*: Erweiterungen eines Datenmodells können dazu führen, dass dieses immer komplexer wird. Dies macht es letztlich schwieriger, mit einem solchen Datenmodell zu arbeiten, dieses zu verstehen und dieses auch entsprechend in Software umzusetzen. Beispielsweise kann die Einführung eines neuen Geometrietyps (z. B. NURBS (Schoenberg, 1964)) dazu führen, dass zukünftig neue Entitäten unterstützt werden müssen, für welche bei den Softwareentwicklern eine entsprechende Expertise vorhanden sein bzw. diese Expertise evtl. erst aufgebaut werden muss. Auch die Erweiterung von IFC in Richtung des Tiefbaus führt dazu, dass für eine vollständige Unterstützung dieses Standards in Zukunft nicht nur schwerpunktmäßig Domänenwissen aus dem Bereich des Hochbaus benötigt wird, sondern zunehmend auch mehr Wissen aus dem Bereich des Tiefbaus. Anzumerken ist hierbei, dass eine vollständige Unterstützung im Allgemeinen nicht notwendig ist und aus diesem Grund das Konzept der Model View Definition (MVD) entwickelt wurde, das es erlaubt, nur eine Untermenge des Datenmodells zu unterstützen. Jedoch liegt eine Motivation für das MVD-Konzept genau darin, die Komplexität des vollumfänglichen Modells zu reduzieren. Eine Verringerung der Komplexität wird im MVD-Konzept durch das Prinzip des Ausschlusses umgesetzt, z. B. durch den Ausschluss von bestimmten Entitäten. Die Komplexitätsreduktion durch Ausschluss stellt aber nicht immer zwangsläufig das optimale Verfahren für eine Komplexitätsreduktion dar. Die Komplexität könnte beispielsweise auch mithilfe von Schnittstel-

len (Interfaces) reduziert werden, was jedoch durch MVDs nicht unterstützt wird.

4.3 Erweiterung der Instanzebene durch Programme

Derzeit in der Praxis verwendete Bauwerksdatenmodelle sind im Regelfall objektorientiert modelliert und unterscheiden zwischen einer Klassenebene (Schemaebene) und einer Objektebene (Instanzebene). Beim Datenaustausch befindet man sich auf Instanzebene. Diese spezifiziert konkrete Entitäten (Objekte) eines Bauwerksmodells. Gegenwärtig fokussiert sich der Datenaustausch dieser Modelle hauptsächlich auf strukturrelevante Daten von Instanzen (konkreten Entitäten bzw. Objekten). Zu den strukturrelevanten Daten eines Objektes gehören der Objekttyp und die Wertebelegung für jedes Strukturmerkmal (Attribute der Entität bzw. Klasse). Die Attribute bilden u. a. auch die Beziehungen (Assoziationen) zwischen den Objekten ab.

Methoden, die auf den strukturrelevanten Daten operieren, oder algorithmisches Wissen, das Berechnungsverfahren definiert, ist auf Instanzebene in den in der Praxis üblichen Datenformaten nicht anzutreffen. Auf Schemaebene wird diese Form des Wissenstransfers zwar teilweise in den gängigen Austauschdatenformaten unterstützt, aber nur sehr eingeschränkt genutzt. IFC kennt hier beispielsweise Derived-Attributes, die es erlauben, auf Basis von Attributwerten und EXPRESS-Programmen Berechnungen anzustellen, um dadurch abgeleitete Attributwerte ermitteln zu können. Jedoch fehlt ein solcher Mechanismus, der es erlaubt, auf eine solche Weise algorithmisches Wissen zu transportieren, komplett auf der Instanzebene. Aus Sicht der Instanzebene ist in Datenformaten wie der IFC nicht vorgesehen, Algorithmen oder Methoden, die durch eine formale Sprache definiert sind und damit einen ausführbaren Programmablauf festlegen, beim Datenaustausch mit zu übergeben.

Dabei hätte jedoch ein solcher Datenaustausch, der den Transfer von Programmen (Methoden bzw. Algorithmen) miteinschließt, eine Reihe von Vorteilen zu bieten (siehe dazu auch Abschnitt 4.4):

- Vermeidung von Standardisierungsarbeit bei Modellerweiterungen, die durch eine Schnittstellendefinition abgedeckt werden können
- Reduzierung des Implementierungsaufwands bei der Unterstützung von Bauwerksdatenmodellen
- Sicherstellung der korrekten Interpretation und fehlerfreien Umsetzung beim Datenaustausch
- Reduzierung der Komplexität von Bauwerksdatenmodellen

Auf Instanzebene ist in gängigen Standards zum Austausch von Bauwerksmodellen im Grunde genommen nur ein Teil des objektorientierten Paradigmas umgesetzt. Dieser Teil berücksichtig nur die statischen Aspekte des objektorientierten

Designs wie Klassenattribute und Assoziationen, nicht jedoch die dynamischen Aspekte, die durch Funktionen oder Methoden beschrieben werden und auf den statischen Attributen operieren können. Bislang fehlt der Austausch von Methoden und Funktionen (Programmen) in etablierten Austauschformaten gänzlich. Der Kernansatz der vorliegenden Arbeit liegt darin, Bauwerksdatenmodelle so zu erweitern, dass diese auch auf Instanzebene Programme austauschen können, um damit den genannten Problemen aus Abschnitt 4.2 entgegenzuwirken und die vorher genannten Vorteile zu erzielen.

Anstatt also nur strukturrelevante Daten beim Datenaustausch zu berücksichtigen, soll es zusätzlich möglich sein, Programme auszutauschen. Ein Programm implementiert dabei auf Basis einer formalen Sprache (Programmiersprache) eine Schnittstelle. Die Schnittstelle ist auf Schemaebene angesiedelt und definiert Methoden, die durch Programme auf Instanzebene implementiert werden.

Abbildung 4.3 stellt die Schemaebene im Überblick dar. Die bisherige typische Verfahrensweise ist durch die Klassen GeometricShape, Rectangle und Circle exemplarisch dargestellt. Diese Entitäten repräsentierten nur strukturrelevante Daten (widht, height, radius). Dem gegenübergestellt ist der hier vorgeschlagene Ansatz. Dieser besteht aus einer Schnittstellenbeschreibung, die beispielhaft durch die Schnittstelle IGeometricShape dargestellt ist. Hierbei wird davon ausgegangen, dass es für verschiedene Anwendungen nur von Bedeutung ist, den Flächeninhalt und den Umkreis einer geometrischen Figur ermitteln zu können. Zudem definiert der Ansatz eine Klasse InterfaceRealizationLink und eine Klasse Program. Die Klasse Program enthält den Quellcode (SourceCode) einer Implementierung einer Schnittstelle (wie z. B. der IGeometricShape-Schnittstelle). Der Quellcode wird mithilfe einer formalen Sprache (Programmiersprache) umgesetzt. Außerdem merkt sich die Klasse mittels des Attributs InterfaceName, welche Schnittstelle (basierend auf deren Namen) durch das Programm umgesetzt wird. Dies ist notwendig, damit eine importierende Anwendung herleiten kann, welches Programmobjekt welche Schnittstelle implementiert. Alternativ könnte diese Information auch aus dem Quelltext des Programmobjekts abgeleitet werden, insofern die formale Sprache, die zur Beschreibung solcher Programme verwendet wird, ein solches Konzept unterstützt. Konzeptionell ist es hier nur wichtig, dass zu einem Program-Objekt ermittelt werden kann, welches Interface dieses implementiert. Die Klasse InterfaceRealizationLink hält einen Link zu einem Program-Objekt und stellt ein Attribut für Initialisierungsdaten bereit. Verschiedene Programminstanzen (Program-Objekte) können auf diese Weise mit unterschiedlichen Daten initialisiert werden.

Auf der Applikationsebene (siehe Abbildung 4.3) muss nun ein Übersetzer (Translator) bereitgestellt werden, der auf Basis eines Program-Objektes und entsprechenden optionalen Initialisierungsdaten ein Objekt erzeugen kann, welches ein vorher definiertes Interface wie IGeometricShape implementiert und dieses entsprechend der Zielanwendung (Anwendung, die die Daten importiert) bereitstellt. Dieser Übersetzer kann als Interpreter oder Compiler realisiert werden (siehe dazu auch Abschnitt 7.2).

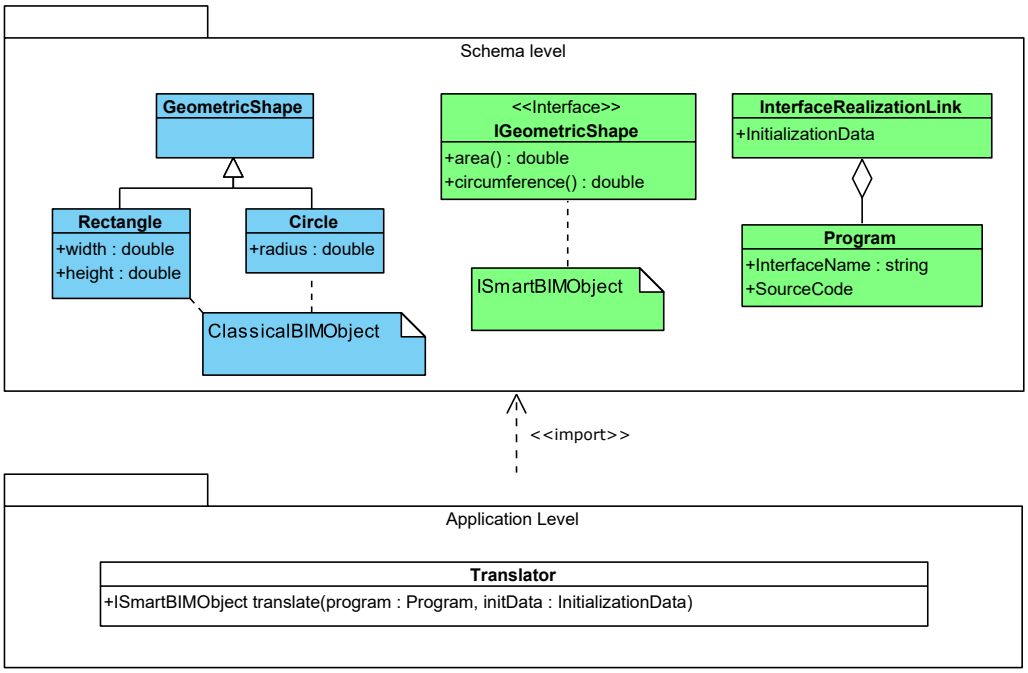


Abbildung 4.3: Schema- und Applikationsebene im Überblick

Abbildung 4.4 stellt die Instanzebene dar. Hierbei werden die Objekte dargestellt, die bei einem Datenaustausch beteiligt sind. Beim konventionellen Ansatz werden nur strukturrelevante Daten ausgetauscht. Dies wird in der Abbildung durch die Instanz der Klasse Rectangle exemplarisch dargestellt. Bei dem hier vorgeschlagenen Ansatz werden Program-Instanzen ausgetauscht. Diese beinhalten die Realisierung einer Schnittstelle der Schemaebene in einer formalen Sprache. Zudem können diese Programme mithilfe einer Instanz der Klasse InterfaceRealizationLink mit Initialisierungswerten versehen werden. Im Kontext des Beispiels der geometrischen Figuren wird es durch den Austausch von Programmen möglich, neue vorher nicht vereinbarte geometrische Figurklassen zu definieren und Instanzen dieser Klassen zwischen verschiedenen Anwendungen auszutauschen. Anwendungen können dabei auf die vereinbarten Schnittstellenmethoden zugreifen, um den Flächeninhalt oder Umfang der entsprechenden Figur zu berechnen, müssen jedoch nicht zwangsläufig die Interna der geometrischen Figuren kennen. Im Kontext des Beispiels der geometrischen Figuren wird es durch den Austausch von Programmen möglich, neue vorher nicht vereinbarte geometrische Figurklassen zu definieren und Instanzen dieser Klassen zwischen verschiedenen Anwendungen auszutauschen. Anwendungen können dabei auf die vereinbarten Schnittstellenmethoden zugreifen, um den Flächeninhalt oder Umfang der entsprechenden Figur zu berechnen, müssen jedoch nicht zwangsläufig die Interna der geometrischen Figuren kennen.

Um den beschriebenen Ansatz zu realisieren, werden folgende Komponenten benötigt:

- eine *Schnittstellenbeschreibung*, die das Verhalten eines Objektes festlegt;

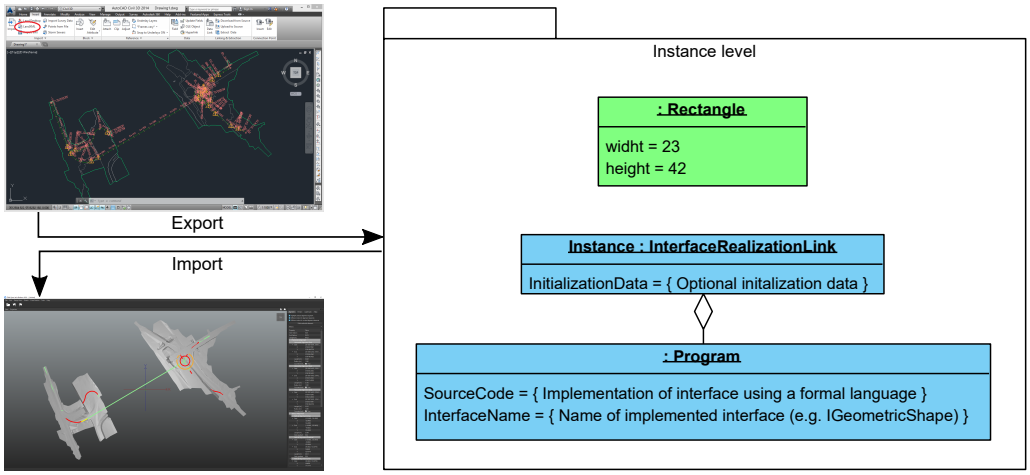


Abbildung 4.4: Instanzebene im Überblick

- eine *formale Sprache*, auf deren Basis Schnittstellenbeschreibungen implementiert werden können;
- ein *Übersetzer*, der ausführbare Realisierungen der implementierten Interfaces auf Basis der formalen Sprache für die Zielapplikation, die diese Daten importieren will, erzeugen kann.

Damit dieses Konzept in den Datenaustauschprozess integriert werden kann, sind zudem Änderungen an den bestehenden Bauwerksdatenmodellen und den Applikationen, die am Datenaustausch beteiligt sind, notwendig. Auf Schemaebene der Bauwerksdatenmodelle müssen Klassen eingeführt werden, die auf Basis einer formalen Sprache implementierte Programme transportieren können. Dieser Aufwand ist relativ gering. Es müssen hier im Wesentlichen nur die Entitäten InterfaceRealizationLink und Program eingeführt werden. Zudem müssen für BIM-Objekte, die diesen Ansatz nutzen wollen, Schnittstellen definiert werden. Dies unterscheidet sich aber nicht vom gängigen Standardisierungsprozess. Dort werden anhand der Anforderungen der Benutzer ebenfalls neue Entitäten eingeführt und zudem explizit definiert, welche Attribute Entitäten bereitstellen müssen. Im Gegensatz dazu wird im vorgeschlagenen Ansatz nicht die konkrete Ausprägung der Attribute festgelegt, sondern eine Schnittstelle definiert, die die Anforderungen der Nutzer widerspiegelt. Dabei wird einerseits eine feste Schnittstelle für die Implementierung festgelegt, anderseits aber nicht explizit definiert, welche Attribute exakt gespeichert werden müssen. Der Aufwand der Schnittstellendefinition ist vergleichbar mit dem Aufwand, der bei der Realisierung neuer traditioneller Entitäten entsteht. Auch wenn Schnittstellendefinitionen keine expliziten Attribute vorschreiben, können Schnittstellenimplementierungen sehr wohl auch auf explizit definierte Attribute zugreifen, falls dies erforderlich sein sollte. Dies wird dadurch erreicht, dass die Initialisierungswerte (InitializationData) eines Programmobjekts um die benötigten expliziten Attribute erweitert werden, bevor diese an die entsprechende Schnittstellenimplementierung weitergereicht werden (siehe dazu auch Abschnitt 8.1.3.3).

Einen großen Aufwand verursacht die Definition einer formalen Sprache zur Übertragung von Programm-Instanzen und die Entwicklung eines Übersetzers für eine solche Sprache. Dieser Aufwand kann geringgehalten werden, indem man eine bereits existierende Sprache wiederverwendet und auf existierende Werkzeuge für den Übersetzerbau zurückgreift. Im Rahmen dieser Arbeit wurden eine formale Sprache sowie eine prototypische Implementierung eines Übersetzers realisiert, die man für diesen Zweck übernehmen kann.

Eine weitere Herausforderung stellt die Integration dieses Ansatzes in eine Applikation dar. Dieser Aufwand kann sich aber amortisieren. Beispielsweise nahm die Unterstützung von Klothoiden in der TUM Open Infra Platform, basierend auf dem IFC 4.1 Standard, in etwa ähnlich viel Zeit in Anspruch wie die Integrierung eines flexiblen Ansatzes für Übergangskurven, der, basierend auf auszutauschenden Programmen, Übergangskurven beschreibt. Wenn man hierbei bedenkt, dass es noch eine ganze Reihe weiterer Übergangskurven gibt, so wird klar, dass sich hier der Aufwand in der Entwicklung durch einen programmbasierten Ansatz sogar reduzieren lässt.

Für die Umsetzung dieses Ansatzes bieten sich verschiedene Realisierungsmöglichkeiten an. In den folgenden Kapiteln wird ausführlich ein Ansatz, der auf dem IFC-Datenmodell basiert, dargestellt.

4.4 Vorteile des vorgestellten Ansatzes

Der vorgestellte Ansatz basiert auf der Definition von wohldefinierten Schnittstellen. Schnittstellen (Interfaces) definieren hierbei einen Vertrag, der festlegt, welche Services (Methoden) durch eine Implementierung bereitgestellt werden müssen. Anwendungen, die den programmbasierten Ansatz unterstützen möchten, müssen nur die durch das Datenmodell vereinbarten Schnittstellen unterstützen und können dadurch losgelöst von konkreten Implementierungen der entsprechenden Schnittstellen umgesetzt werden. Dadurch ist es möglich, Software gegen eine Schnittstelle zu entwickeln und nicht, wie bisher üblich, gegen eine konkrete Realisierung, die in Form von Entitäten und Attributen in einem Datenaustauschstandard spezifiziert ist. Dieses Prinzip „Program to an interface, not an implementation“ wird unter anderem in (Gamma *et al.*, 1995) beschrieben und ist ein grundlegendes Prinzip des objektorientierten Designs. Eine Anwendung kann davon ausgehen, dass eine Realisierung einer Schnittstelle den definierten Vertrag einhält und alle Services wie vereinbart zur Verfügung stellt. Dabei spielt es für eine Anwendung keine Rolle, wie eine Realisierung einer Schnittstelle im Detail umgesetzt wurde. Dies kann in einigen Situationen helfen, einen zeit- und kostenintensiven Standardisierungsprozess zu vermeiden. So könnte beispielsweise eine Schnittstelle für Übergangskurven standardisiert werden, die genutzt werden kann, um beliebige Realisierungen von Übergangskurven umzusetzen. Konkrete Realisierungen der Schnittstellen könnten z. B. Klothoiden oder Bloss-Bögen sein. Deren Umsetzung müsste dabei nicht standardisiert werden, sondern diese müssen nur die vereinbarte Schnittstelle für Übergangskurven umsetzen. Auf diese Weise könnten auch noch weitere Übergangskurventypen unterstützt werden, oh-

ne tatsächlich das Datenmodell zu erweitern bzw. ohne eine erneutes Standardisierungsverfahren in Gang setzen zu müssen. Durch wohldefinierte Schnittstellen müssen Erweiterungen des Datenmodells nicht mehr auf der Ebene von Entitäten und Attributen spezifiziert werden, was im Regelfall einen langen Standardisierungsprozess erfordert, sondern können auf Basis einer abstrakten Schnittstellenbeschreibung einfach in das Datenmodell ohne einen aufwändigen Standardisierungsprozess mittels einer Interface-Implementierung integriert werden. Dies stellt einen Vorteil des programmbasierten Ansatz dar.

Unterstützt eine Software den programmbasierten Ansatz, kann diese mit jeder Realisierung eines Interfaces arbeiten, ohne dass dafür ein zusätzlicher Entwicklungsaufwand für ein Softwarehaus entsteht. Ohne den vorgestellten Ansatz würde jede Erweiterung, die durch die Realisierung eines Interfaces umgesetzt werden könnte, mittels klassischen Entitäten und Attributen abgebildet werden, die dann jedes Softwarehaus für sich eigens verstehen, implementieren, testen und deployen müsste. Der Interface-basierte Ansatz kann diesen langwierigen Softwareentwicklungsprozess verkürzen und damit helfen, dass Erweiterungen des Datenmodells schneller durch Softwareprodukte unterstützt werden können. Im Falle des Beispiels mit den Übergangkurven ist es vorstellbar, dass eine Sinusoide als Übergangsbogen unterstützt werden soll. Im klassischen Ansatz ohne programmbasierten Ansatz würde man dazu das zugrundeliegende Datenmodell entsprechend anpassen, so dass Sinusoiden als Übergangskurven unterstützt werden können. Diese Anpassung des Datenschemas erfordert jedoch auch eine entsprechende Anpassung der Software, die das um Sinusoiden erweiterte Datenschema verarbeiten kann. Dabei muss ein Softwarehaus die Details wie z. B. die verwendeten Parameter zur Beschreibung der Sinusoide entsprechend softwaretechnisch unterstützen, was mit einem gewissen Entwicklungsaufwand verbunden ist. Der programmbasierte Ansatz hilft, diesen Entwicklungsaufwand zu vermeiden. Im programmbasierten Ansatz muss eine Anwendung nur eine Schnittstellenbeschreibung unterstützen und nicht jede Realisierungsvariante davon. Eine Software, welche eine Schnittstellendefinition für Übergangskurven unterstützt, kann implizit mit jeder Realisierungsvariante umgehen und muss diese nicht explizit implementieren.

Beim konventionellen Datenaustausch stellt sich ein weiteres Problem: Ein Standard muss richtig verstanden und korrekt umgesetzt werden. Durch einen programmbasierten Datenaustausch werden solche Fehlerquellen minimiert. Entwickler müssen nicht die Details einer Variante einer fremden Schnittstellenimplementierung verstehen und diese auch nicht selbst implementieren, sondern können direkt die Implementierung einer Schnittstelle eines fremden Anbieters verwenden. Dadurch können beispielsweise Softwarefehler vermieden werden, die durch Fehlinterpretation einer informalen Dokumentation entstehen können. Im Kontext des Beispiels der Übergangskurven kann hier die Implementierung einer Sinusoide von einer Anwendung wiederverwendet werden und muss nicht neu implementiert werden. Dadurch kann vermieden werden, dass beispielsweise die Daten der Sinusoide falsch interpretiert oder fehlerhaft (anders als von der Quellapplikation) interpretiert werden.

Die Komplexität eines Bauwerksdatenmodells nimmt mit der Einführung und Verwendung zusätzlicher Entitäten und Attribute zu. Der programmbasierte Ansatz hilft durch die Auslagerung von Entitäten und Attributen in Schnittstellenerweiterungen, die Komplexität eines Datenschemas nicht noch weiter zu erhöhen. Am Beispiel der Übergangkurven können Implementierungsdetails wie die Parameter, die zur Beschreibung von Übergangskurven verwendet werden, in einer Schnittstellenimplementierung ausgelagert werden. Dadurch kann vermieden werden, dass die Komplexität eines Datenschemas erhöht wird. Dies wird durch eine klare Trennung zwischen Schnittstelle und Implementierung erzielt.

Eine Schemaerweiterung im Rahmen der IFC ist, wie bereits diskutiert worden ist, immer mit einem erheblichen Aufwand verbunden. Daher wird im Rahmen der IFC intensiv Gebrauch von Property-Sets für nationale bzw. projektspezifische Erweiterungen gemacht. Dies reduziert allerdings die semantische Kohärenz, da eindeutige Definitionen der Bedeutung der verwendeten Eigenschaftswerte (Properties) häufig nicht existieren. Wird die Interpretation in Form eines Programms mitgeliefert, wie durch den programmbasierten Ansatz vorgeschlagen, entfällt dieses Problem.

4.5 Zusammenfassung

In diesem Kapitel wurde erläutert, wie nach heutigem Stand der Technik prinzipiell der herstellerneutrale Datenaustausch von Bauwerksdatenmodellen betrieben wird. Es wurde auf Probleme eingegangen, die derzeit bei der Umsetzung eines plattformneutralen Datenaustausches von Bauwerksdatenmodellen bestehen. Dabei wurden Standardisierungsprozesse, die Adaptierung von Standards sowie die Zunahme der Komplexität von Bauwerksdatenmodellen als Problemquellen identifiziert, die dazu führen, dass die Etablierung von offenen Standards zeit- und kostenintensiv ist. Um diesen Nachteilen entgegenzuwirken, wurde ein Konzept vorgestellt, das den programmbasierten Datenaustausch in den Fokus stellt. Programme implementieren dabei auf Basis einer formalen Sprache vordefinierte Schnittstellen. Dadurch können Implementierungsdetails aus dem Standardisierungsprozess extrahiert, die Adaptierung von Standards beschleunigt, Softwarefehler vermieden und die Komplexität von Bauwerksdatenmodellen reduziert werden.

Kapitel 5

Die IFC Programming Language

Die IFC-Programmiersprache (IFC Programming Language oder kurz IFC-PL) erlaubt es, Programme (Algorithmen) zu verfassen und auf Basis der IFC in einem herstellerneutralen Datenformat auszutauschen. Die definierten Algorithmen können beispielsweise Verarbeitungs- oder Analyseprozesse beinhalten. Mögliche Anwendungen für die IFC-PL reichen von der Definition komplexer Funktionen für die Mengenermittlung bis zur Beschreibung des „Verhaltens“ von parametrischen Objekten (Lee *et al.*, 2006). Da diese Programme einen Teil des Austauschprozesses darstellen, können diese wiederum bei der Empfängerseite (Softwareapplikation) genutzt werden. Auf diese Weise können der Programmieraufwand auf der empfangenden Seite reduziert und Fehlinterpretationen der Daten verhindert werden. Betrachtet man aktuelle Softwareprodukte, die IFC-Dateien austauschen, sind Datenverluste und Fehlerinterpretationen leider häufig anzutreffen. Genauso werden häufig Standards nicht vollständig implementiert, da es schlichtweg zu zeitaufwändig für die Softwarehersteller ist, jedes Detail des Datenmodells zu unterstützen. Dies rechtfertigt den hier gewählten Ansatz.

Ein IFC-PL-Programm selbst ist nicht Teil des IFC-Schemas, sondern Teil des Datenaustausches. Neben einer IFC-Instanzdatei wird auch der Quelltext in UTF-8-Form (Pavel *et al.*, 2016) des IFC-PL-Programmes weitergereicht. IFC-PL-Programme sollten mit der Endung .ifcpl versehen werden. IFC-PL-Programme befinden sich nicht auf der Schemaebene, sondern auf Instanzebene. Im Vergleich zu einem Ansatz auf Schemaniveau ermöglicht der instanzbasierte Ansatz eine wesentlich höhere Flexibilität in Bezug auf das Definieren von neuen IFC-PL-Programmen, da eine Schemamodifikation einen langwierigen Standardisierungsprozess nach sich ziehen würde. Trotzdem müssen IFC-PL-Programme für eine bestimmte Schnittstelle definiert werden. Die Interfacedefinition kann ebenfalls mithilfe der IFC-PL umgesetzt werden.

IFC-PL verwendet sehr ähnliche Sprachkonzepte wie die Programmiersprachen C++, C# und Java. Der Umgang mit Variablen, Kontrollstrukturen und Klassen funktioniert dort in sehr ähnlicher Weise.

Die IFC-PL wurde im Rahmen dieser Arbeit entwickelt und wird im Weiteren genauer beschrieben.

5.1 Grundsätzliche Überlegungen zum Design der Sprache

IFC-PL soll den einfachen Umgang mit den Typen und Entitäten eines EXPRESS-basierten Datenmodells ermöglichen. Das heißt, es soll für den Anwender der Sprache möglich sein, auf einfache Weise Instanzen, basierend auf Entitäten eines EXPRESS-Schemas, zu erstellen, diese zu verarbeiten und sie entsprechend serialisieren und deserialisieren zu können. Dabei soll jede Entität mit all ihren Attributen direkt innerhalb eines IFC-PL-Programms abgebildet werden und verwendbar sein. Gleichzeitig soll dadurch eine Typprüfung umgesetzt werden, die gewährleistet, dass nicht auf Attribute zugegriffen werden kann, die nicht vorhanden sind, oder Attribute mit Daten des falschen Typs beschrieben werden. Dies erfordert, dass die Sprache streng typisiert sein muss.

Aufgrund der inhärenten objektorientierten Natur des EXPRESS-Datenmodells soll auch die IFC-PL das objektorientierte Paradigma unterstützten. Es soll Klassen mit Attributen und Vererbungsbeziehungen geben. Dabei soll IFC-PL die Grundfunktionalitäten einer objektorientierten Programmiersprache bereitstellen.

Außerdem sollen auf Basis der IFC-PL Schnittstellen und Software-Module implementiert werden, die bestimmte bauwerksrelevante Informationen verarbeiten können. Diese Software-Module (Programme) sollen zwischen verschiedenen Fachanwendungen ausgetauscht und verwendet werden können. Dabei soll es an der Schnittstelle zwischen zwei Fachapplikationen möglich sein, EXPRESS-basierte Typen des zugrundeliegenden EXPRESS-Schemas auszutauschen.

Die Basis zur Formulierung dieser Programme soll ein prozeduraler Sprachkern bilden. Dies macht Sinn, da EXPRESS Typen für Variablen definiert und das prozedurale Paradigma sich gerade dadurch auszeichnet, dass es den Zustand eines Systems mit Variablen beschreibt. Es bietet sich daher darüber hinaus auch an, die Formulierung algorithmischer Systemabläufe (Sequenzen, Schleifen, Bedingungen) ebenfalls auf die Basis der prozeduralen Programmierung zu stellen.

IFPCL soll für Programmierer, die bereits in gängigen objektorientierten Sprachen Erfahrungen gesammelt haben, eine relativ leicht lern- und anwendbare Hochsprache sein. Daher macht es Sinn, Konzepte aus anderen objektorientierten Programmiersprachen zu adaptieren.

Die Anforderungen, die an die IFC-PL gestellt werden, sind zusammengefasst:
- eine streng typisierte Umgebung

- objektorientierte Konzepte
- prozeduraler Sprachkern
- bruchlose Integration von EXPRESS-basierten Typen
- leichte Lern- und Anwendbarkeit durch Orientierung an bestehenden objektorientierten Sprachkonzepten

Prinzpell lassen sich diese Anforderungen mit einem Early-Binding-Ansatz auf Basis einer existierenden Programmiersprache wie C++, C# oder Java realisieren, haben jedoch im Vergleich zum IFC-PL-Ansatz einige Nachteile:

– Der Standalone-Nachteil: Die Verwendung eines Early-Binding-Generators erfordert zusätzlichen Aufwand auf Seiten des Entwicklers. IFC-PL soll diesen manuellen Aufwand für den Entwickler vermeiden.

– Ein Early-Binding-Generator erzeugt *boilerplate code*: Auf Basis der IFC-PL sollen Software-Module ausgetauscht werden. Der durch eine Early-Binding erzeugte Code wäre Teil dieser Software-Module und müsste mitausgetauscht werden. oipEpress, ein Early-Binding-Generator für C++, erzeugt für das IFC 4.1-Binding 141601 Zeilen Code. Dieser Code müsste bei einem Dateiaustausch mit berücksichtig werden. IFC-PL stellt ein Sprachkonzept bereit (siehe Imported Types Abschnitt 5.20), das diesen Codeaustausch auf eine einzige Zeile Code reduziert.

  Würde man für Software-Module eine Sprache wie C++, C# oder Java einsetzen, so hätte man bei der Definition diese Datenaustausches nicht direkt konzeptionell mit der ausgetauschten Entität des EXPRESS-Schemas selbst zu tun, sondern mit einer generierten Entsprechung dieser Entität in der zugehörigen Programmiersprache. Dies würde zwar prinzipiell auch funktionieren, hätte aber beispielsweise bei Schnittstellenbeschreibungen zur Folge, dass diese aufgrund von unzulänglichen Sprachmitteln mit *boilerplate code* gespickt wären, wie folgendes Beispiel zeigt:

```
1   #include "IFC4x1/include/IfcGloballyUniqueId.h"
2   #include "IFC4x1/include/IfcIdentifier.h"
3   #include "IFC4x1/include/IfcLabel.h"
4   #include "IFC4x1/include/IfcObjectPlacement.h"
5   #include "IFC4x1/include/IfcOwnerHistory.h"
6   #include "IFC4x1/include/IfcProductRepresentation.h"
7   #include "IFC4x1/include/IfcRelAggregates.h"
8   #include "IFC4x1/include/IfcRelAssigns.h"
9   #include "IFC4x1/include/IfcRelAssignsToProduct.h"
10  #include "IFC4x1/include/IfcRelAssociates.h"
11  #include "IFC4x1/include/IfcRelConnectsElements.h"
12  #include "IFC4x1/include/IfcRelConnectsWithRealizingElements.h"
13  #include "IFC4x1/include/IfcRelContainedInSpatialStructure.h"
14  #include "IFC4x1/include/IfcRelDeclares.h"
15  #include "IFC4x1/include/IfcRelDefinesByObject.h"
16  #include "IFC4x1/include/IfcRelDefinesByProperties.h"
17  #include "IFC4x1/include/IfcRelDefinesByType.h"
```

```
18    #include "IFC4x1/include/IfcRelFillsElement.h"
19    #include "IFC4x1/include/IfcRelInterferesElements.h"
20    #include "IFC4x1/include/IfcRelNests.h"
21    #include "IFC4x1/include/IfcRelProjectsElement.h"
22    #include "IFC4x1/include/IfcRelReferencedInSpatialStructure.h"
23    #include "IFC4x1/include/IfcRelSpaceBoundary.h"
24    #include "IFC4x1/include/IfcRelVoidsElement.h"
25    #include "IFC4x1/include/IfcText.h"
26    [...]
27
28    class IMyInterface {
29        virutal IfcRepresentation
30        getRepresentation(const IfcPropertySet& ps) = 0;
31        [...]
32    }
```

IFC-PL ist eine Hochsprache, die sich stark an den Grundzügen der Syntax von etablierten objektorientierten Mehrzwecksprachen wie C++ (ISO, 2014), C# (ECMA International, 2006) oder Java (Gosling *et al.*, 2014) orientiert. Darüber hinaus bietet die IFC-PL domänenspezifische Erweiterungen für den Zugriff auf EXPRESS-basierte Datenmodelle an. Die Zielgruppe der Programmiersprache stellen Softwareentwickler dar, die Erfahrungen mit EXPRESS-basierten Datenmodellen und objektorientierten Sprachen aus der C++-verwandten Sprachfamilie besitzen. Nicht zur Zielgruppe gehören Personen, die keine Erfahrung im Umgang mit dieser Art von Datenmodellen bzw. Programmiersprachen haben.

Laut dem TIOBE-Index (Stand März 2018) (TIOBE-Software-BV, 2018a), der versucht, die Popularität von Programmiersprachen widerzuspiegeln, rangieren Java, C++ und C# neben C unter den zehn beliebtesten Programmiersprachen. Der TOIBE-Index berechnet sich auf Grundlage der Trefferanzahl bei Suchmaschinen zu der Suchanfrage `+"<language> programming"`. Dabei ist der Begriff der Suchmaschine in TOIBE so definiert, dass dieser nicht nur klassische Suchmaschinen miteinbezieht, z. B. Google oder Bing, sondern auch Portale, die eine Suchfunktion bereitstellen wie beispielsweise YouTube oder Ebay. Insgesamt stützt sich der Index auf 25 verschiedene Suchmaschinen. Als Programmiersprachen betrachtet der Index nur Sprachen, die einen eigenen Wikipedia-Eintrag besitzen, Turing-vollständig sind und mindestens 5000 Treffer bei Google für die Suchabfrage `+"<language> programming"` erzielen. Weitere Details zur Ermittlung des Indexes sind in (TIOBE-Software-BV, 2018b) zu finden. Auch vergleichbare Rankings, die versuchen, die Popularität einer Programmiersprache zu messen, kommen zu ähnlichen Ergebnissen. Die genannten Programmiersprachen sind beispielsweise auch unter den Top 10 des RedMonk-Indexes (The RedMonk Programming Language Rankings: January 2016) und des PYPL PopularitY of Programming Language (Stand Januar 2017) zu finden. Der Anreiz für IFC-PL, eine ähnliche Syntax zu verwenden, liegt darin, dass der Anwender diese dadurch schneller lernen und verstehen kann, sofern dieser die Vorbilder kennt und nicht von Grund auf eine neue Syntax lernen muss. Die Entscheidung, eine eigene Programmiersprache zu entwerfen, anstatt eine bestehende zu verwenden,

wurde getroffen, um die Anbindung zwischen der Beschreibung des Bauwerksinformationsmodells und der Programmiersprache zu verbessern. In weiten Teilen wurde auf bestehende Konzepte existierender Programmiersprachen gesetzt und spezifische Erweiterungen für den Umgang mit EXPRESS-basierten Bauwerksinformationsmodellen wurden umgesetzt.

Ferner ist es ein Ziel der IFC-PL, eine ähnliche Mächtigkeit und Eleganz für die Formulierung von Programmen zu ermöglichen, wie dies in den C++-verwandten Sprachen möglich ist. Insbesondere werden die Konzepte Kapselung, Vererbung und Polymorphie sowie ihre syntaktische Umsetzung in ähnlicher Weise von IFC-PL adaptiert. Aber auch wichtige fundamentale Konzepte wie z. B. die Flusskontrolle (Schleifen, bedingte Anweisungen, usw.) oder etwa Variablendeklarationen wurden weitgehend übernommen. Auf höherwertige Konzepte wie z. B. umfangreiche Unterstützung für Template-Meta-Programmierung (Alexandrescu, 2001; Vandevoorde & Josuttis, 2002), Delegates und Events (Albahari & Albahari, 2015), Properties (Get- und Setter für Eigenschaften), oder beispielsweise Lambda-Ausdrücke wurde verzichtet. Es wurde versucht, die Konzepte der Kernsprache, d. h. die grundlegenden prozeduralen und objektorientierten Konzepte, die in C++, Java und C# vorhanden sind und von allen drei Sprachen in ähnlicher Weise unterstützt werden, zu übernehmen. Spezielle Entwicklungen, die nur von einer Sprache unterstützt werden (wie ein mächtiger Meta-Programmierungs-Mechanismus aus C++ oder Delegates, Events oder Properties in der Sprache C#) wurden nicht übernommen. Der Grund hierfür liegt darin, dass IFC-PL für Programmierer, die Erfahrung mit C++, Java oder C# haben, leicht zugänglich sein und nicht vorausgesetzt werden soll, dass jeder Programmierer mit jedem Detail jeder dieser Sprachen vertraut ist. Ein anderer Grund ist rein praktischer Natur: Würde man jedes dieser Sprachfeature unterstützen, würde schlichtweg der Aufwand für die Implementierung eins IFC-PL-Übersetzers steigen.

Die Grammatik der IFC-PL hat bei weitem nicht den Umfang der Grammatik von C#, Java oder C++, sondern liefert im Wesentlichen nur grundlegende iterative und objektorientierte Grundmechanismen, die sich stark an die genannten Vorbildsprachen anlehnen. Funktionalitäten wie etwa bitweise Verschiebungsoperatoren (Bit-Shift-Operatoren) wurden nicht umgesetzt. Ebenfalls wurden an manchen Konzepten nur starke Vereinfachungen realisiert. Beispielsweise gibt es nur die syntaktische Möglichkeit, Exceptions zu werfen, aber keinen Mechanismus, um diese zu fangen.

Die Schlüsselwörter der Sprache sind im Anhang B.1 definiert. Die Sprache ist formal im Anhang B.2 in Backus-Naur-Form bzw. mithilfe von regulären Ausdrücken definiert. Im Rahmen dieses Kapitels werden einige informelle Regeln in Textform definiert, die nicht formal durch die Grammatik abgedeckt werden. Diese sind ebenfalls bei einer Umsetzung eines IFC-PL-Übersetzers zu berücksichtigen. Informelle Regeln sind fehleranfälliger als formale, jedoch muss hier ein sinnvoller Kompromiss zwischen der Realisierbarkeit und Implementierbarkeit eine IFC-PL-Übersetzers gefunden werden.

Im Folgenden soll anhand von verschiedenen Beispielen die Funktionsweise der IFC-PL verdeutlicht werden.

5.2 Sequentielle Anweisungen

Ein IFC-PL-Programm setzt sich aus Anweisungen (Statements) zusammen.

⟨*program*⟩ ::= ⟨*stmts*⟩

⟨*stmts*⟩ ::= ⟨*stmt*⟩
| ⟨*stmts*⟩ ⟨*stmt*⟩

Eine sequentielle Anweisung kann beispielsweise ein Ausdruck (Expression) sein. Im Folgenden sind drei einfache Statements gezeigt, die auf Expressions basieren:

```
1 5 + 5;
2 3 * 3;
3 4 / 2 - 1;
```

Nach jeder Expressionanweisung steht ein Semikolon (;). Dieses schließt das Expression-Statement ab. Dies gilt auch für andere sequentielle Anweisungen wie z. B. return-, break- oder continue-Anweisungen oder etwa auch für Variablendeklarationen. Statements werden sequentiell entsprechend ihrer Reihenfolge mit der ersten Anweisung beginnend ausgeführt.

Der Unterschied zwischen einem Statement und einer Expression ist, dass eine Expression immer zu einem Wert evaluiert werden kann (Beispiel: `5 + 3 * 9`), wohingegen eine Anweisung nur etwas bewirken muss, aber nicht zwangsläufig zu einem Wert evaluiert werden kann (Beispiel `break`). Expressions sind somit auch Statements, was aber anders herum nicht gilt. Der besseren Übersichtlichkeit halber sollte man jede Anweisung in eine eigene Zeile schreiben, was aber aus Syntaxsicht nicht unbedingt notwendig ist.

Das Verhalten sequentieller Anweisungen folgt, wie der gesamte prozedurale Basiskern, den Vorbildsprachen C++, Java und C#. IFC-PL unterstützt auch die aus diesen Sprachen bekannten Blöcke.

```
1 <block> ::= '\{' <stmts> '\}'
2 \alt '\{' '\}'
```

Ein Block beginnt und endet mit einer geschweiften Klammer. Ein Block ist selbst ein Statement. Dadurch ist auch folgende Programmstruktur möglich:

```
1 5 + 5;
2 {
3     3 * 3;
4
5     {
6         4 / 2 - 1;
```

```
7     }
8 }
```

IFC-PL ist eine *free-form* Language, das heißt, die Art der Einrückungen und Zeilenumbrüche spielt keine Rolle. Dies unterscheidet IFC-PL z. B. von einer Programmiersprache wie Python, bei der Einrückungen eine wichtige Rolle spielen (Zelle, 2010).

5.3 Betriebsmodi

Programme, die in IFC-PL geschrieben werden, können entweder allein stehen, das heißt im Standalone-Modus betrieben, oder alternativ als Bibliothek für andere Programme zur Verfügung gestellt werden. Für den Standalone-Modus muss ein IFC-PL-Programm eine `main`-Funktion bereitstellen[1]. Entdeckt der IFC-PL-Compiler eine `main`-Funktion, so erzeugt er eine ausführbare Datei (unter Windows eine EXE-Datei). Andernfalls wird eine Bibliotheksdatei erzeugt (unter Windows eine Dynamic Link Library (DLL) bzw. unter unixähnlichen Systemen eine Shared Library).

5.4 Hallo-Welt-Programm

Als typisches Standardbeispiel zur Einführung einer Programmiersprache dient im Regelfall ein Hallo-Welt-Programm. Ein IFC-PL-Hallo-Welt-Programm wird im folgenden Listing gezeigt:

```
1 module HelloWorld;
2
3 import Core;
4
5 void main() {
6     print("hello, world\n");
7 }
```

Das Programm beginnt in Zeile 1 mit einer `module`-Anweisung. IFC-PL erlaubt das Aufteilen von Softwarekomponenten in Module. Das Modulsystem von IFC-PL ist in Anlehnung an den Vorschlag der C++-Erweiterung um ein Modulsystem, das in (Reis & Hall, 2014) beschrieben wird, entwickelt worden. Jedes Modul hat dabei einen eindeutigen Namen. In dem gezeigten Beispiel lautet der Modulname HelloWorld. Eine Quelltextdatei muss keine module-Anweisung enthalten und darf höchstens eine solche Anweisung enthalten. Fehlt die Anweisung, wird implizit das Hauptmodul main verwendet.

In Zeile 3 befindet sich eine import-Anweisung, welche das Modul Core importiert. Ein Modul ist eine für sich abgeschlossene Komponente, die eine beliebige Anzahl von Klassen und Funktionen enthalten kann. Um eine Klasse oder Funktion aus

[1] Funktionen werden näher im Abschnitt 5.9 behandelt.

einem Modul verwenden zu können, muss das entsprechende Modul importiert werden. IFC-PL stellt eine Standardbibliothek von Modulen bereit, die häufig benötigte Funktionalitäten beinhalten (siehe Abschnitt 5.19) Ein Modul selbst kann sich über mehrere Quelltextdateien erstrecken. Dabei muss jede Quelltextdatei eines Moduls die Modulzugehörigkeit mittels der module-Anweisung bekannt machen. Quelltextdateien, die IFC-PL-Programmcode enthalten, sollten aus Konsistenzgründen mit der Endung (.ifcpl) versehen werden.

Der Haupteinstiegspunkt wird durch die main-Funktion in Zeile 5 definiert. Eine main-Funktion ist in einem IFC-PL-Programm nicht zwangsläufig nötig. In einem IFC-PL-Programm darf höchstens eine main-Funktion vorkommen. Diese dient, ähnlich wie in einem C- oder C++-Programm, als Einstiegspunkt bei der Ausführung eines IFC-PL-Programms.

Durch die print-Funktion in Zeile 6 wird der String "hello, world" ausgegeben. Die Escape-Sequenz \n wird als Zeilenumbruch interpretiert. Die Definition von Escape-Sequenzen wurde entsprechend von der C/C++-Sprachfamilie übernommen. Siehe hierzu z. B. den C#-Sprachstandard (ECMA International, 2006). Die print-Funktion ist Teil des Modules Core und dient u. a. für Ausgaben auf die Standardausgabe (z. B. Konsolen-/Terminalfenster).

Der Syntaxbaum des Programms ist in den Abbildungen 5.1, 5.2 und 5.3 dargestellt. Die vollständige Grammatik der IFC-PL in Backus-Naur-Form befindet sich im Anhang B.2. Grundlagen zu Syntaxbäumen, Bachus-Naur-Form und Ableitungen werden u. a. in (Vossen & Witt, 2006) beschrieben. Zur besseren Übersichtlichkeit wurde der Syntaxbaum in verschiedene Teilbäume aufgeteilt. Das Startsymbol program befindet sich im Teilbaum, der in Abbildung 5.1 dargestellt ist. Hierbei werden alle Symbole soweit wie möglich aufgelöst, bis auf das Symbol func_decl. Dieses ist im Teilbaum, der in Abbildung 5.2 dargestellt ist, weiter aufgelöst. Auch hier wurde wieder zugunsten der besseren Übersichtlichkeit ein Teilsymbol (block) nicht vollständig aufgelöst, sondern in einer weiteren Abbildung (Abbildung 5.3) dargestellt.

5.5 Case-sensitivity

Der Quelltext von IFC-PL-Programmen ist case-sensitive, d. h. die Groß- bzw. Kleinschreibung von Programmierbefehlen spielt eine entscheidende Rolle. Beispielsweise handelt es sich bei den Variablenbezeichnern myProperty, MyProperty und myproperty um drei unterschiedliche Namen, die drei unterschiedliche Variablen bezeichnen. Sprachen wie C++, Java und C# sind ebenfalls case-sensitive. Bei Visual Basic .NET und einigen älteren Programmiersprachen wie Fortran oder COBOL spielt die Groß- bzw. Kleinschreibung keine Rolle. Die Wahl für Case-sensitivity wurde getroffen, um eine möglichst hohe Ähnlichkeit zu erreichen und gleichzeitig die Kompatibilität zu den Vorbildsprachen zu wahren. Dies ermöglich es z. B., mit geringen Änderungen Quelltext aus C++-, Java- oder C#-Programmen übernehmen zu können.

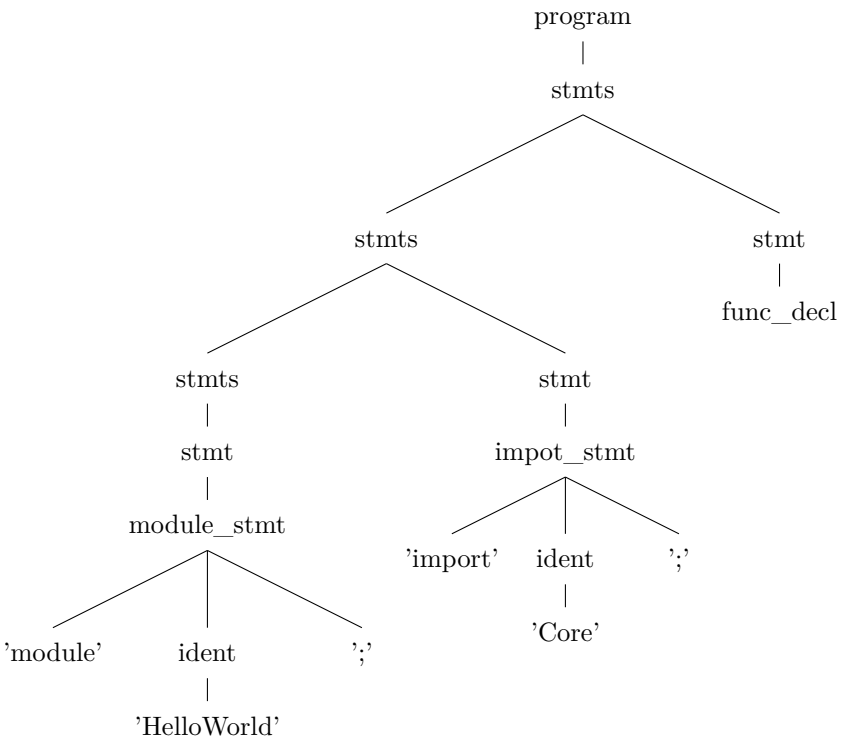


Abbildung 5.1: Das Startsymbol der Grammatik lautet program. Die module- und import-Anweisung sind bis zu ihren Endterminalen aufgelöst. Das Symbol func_decl wird weiter in Abbildung 5.2 aufgelöst.

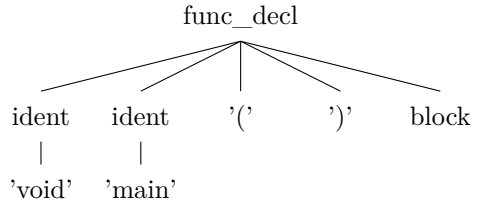


Abbildung 5.2: Die main-Funktion wird als Ableitung des Symboles func_decl beschrieben. Alle Symbole sind bis zu ihren Endterminalen aufgelöst mit Ausnahme des Symbols block. Dieses wird in Abbildung 5.3 weiter aufgelöst.

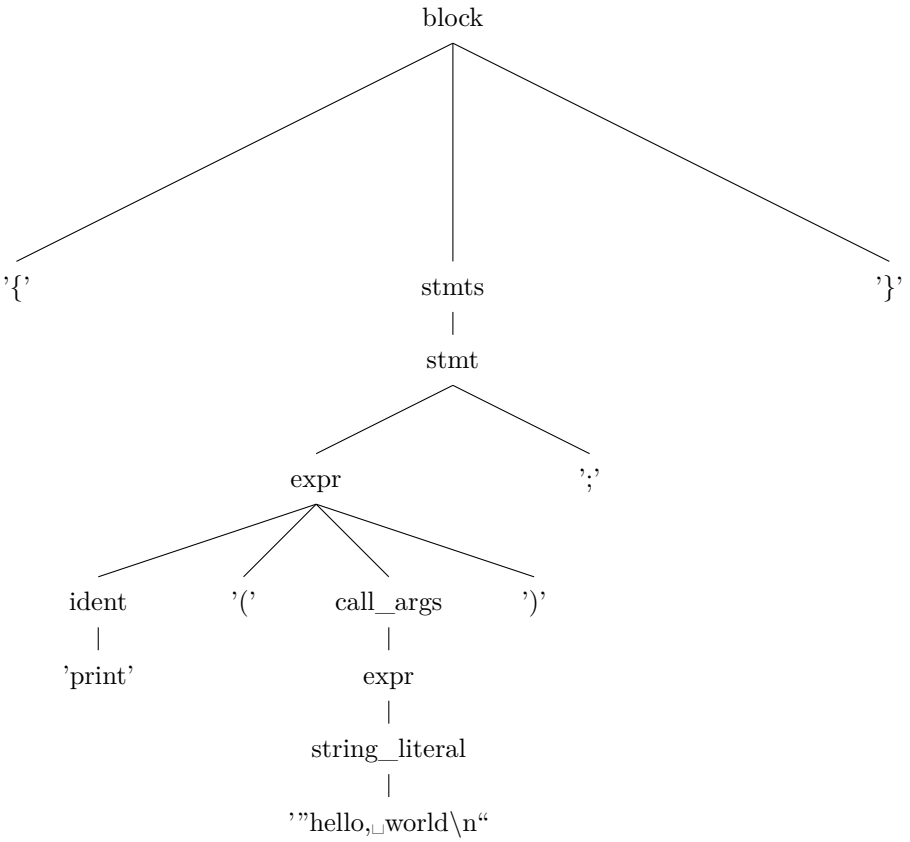


Abbildung 5.3: Der Funktionskörper der main-Funktion wird durch ein Ableitung des Symbols block beschrieben.

5.6 Kommentare

IFC-PL kennt den Zeilenkommentar (// comment) und den Mehrzeilenkommentar (/* comment */). Alle Zeichen innerhalb einer Zeile ab der Zeichenfolge // werden als Kommentar betrachtet. Der Mehrzeilenkommentar kann sich im Gegensatz zum Zeilenkommentar über mehrere Zeilen erstrecken. Dieser wird mit der Zeichenfolge /* eingeleitet und erstreckt sich bis zum Auftreten der Zeichenfolge */, die den Mehrzeilenkommentar abschließt.

```
1 // Das ist ein Zeilenkommentar
2 /* Das ist ein Mehrzeilenkommentar,
3    der sich über mehrere Zeilen erstrecken darf. */
```

5.7 Variablentypen

IFC-PL ist eine streng typisierte Sprache. Daher muss für jede Variable auch ein Datentyp festgelegt werden. Tabelle 5.1 zeigt einen Überblick über die von IFC-PL unterstützten primitiven Datentypen.

Primitiver IFC-PL-Datentyp	**Defaultwert**
bool	false
logical	false
char	'\0'
short	0
int	0
long	0
float	0
double	0

Tabelle 5.1: Default-Werte von primitiven IFC-PL-Datentypen

Für boolesche Werte (false, true) steht der Datentyp bool bzw. logical bereit. Dabei kennt der Datentyp logical neben den Zuständen false und true zusätzlich noch den Zustand unknown, der einem unbekannten logischen Zustand entspricht. Der Datentyp char besitzt eine Größe von 8 Bits (1 Byte) und ist mittels UTF-8 (8-Bit Universal Character Set Transformation Format) kodiert. Sollen Zeichen kodiert werden, die außerhalb der Ein-Byte-Grenze liegen wie z. B. der Umlaut 'ö', muss dieses mittels eines Feldes (Arrays) vom Datentyp char umgesetzt werden. Als Datentypen für Ganzzahlen können die Typen short, int und long genutzt werden. Diese haben eine entsprechende Größe von jeweils 16-, 32- und 64-Bit. Für Fließkommazahlen stehen die Datentypen float und double bereit, die jeweils 32- bzw. 64-Bit umfassen. Primitive Datentypen werden mit einem Default-Wert initialisiert. Tabelle 5.1 gibt einen Überblick über die Default-Werte für primitive Datentypen.

Bei den primitiven Datentypen der IFC-PL handelt es sich um sogenannte Wertetypen (value types). Das heißt, dass bei einer Übergabe an eine Funktion oder Methode der Wert einer Variablen eines primitiven Datentyps kopiert bzw. übergeben wird, jedoch nicht seine Speicheradresse bzw. die entsprechende Referenz. Dies hat zur Folge, dass eine Änderung des kopierten Werts keine Veränderung des Werts der kopierten Variable zur Folge hat.

Für die numerischen Datentypen werden entsprechend unäre- und binäre-Operatoren unterstützt, um mathematische Operationen wie z. B. eine Addition oder eine Multiplikation durchführen zu können.

Weiter wird der Variablentyp string bereitgestellt, um grundlegende Zeichenkettenverarbeitung zu ermöglichen. Bei einer Variable vom Typ string handelt es sich, anders als bei den primitiven Datentypen, nicht um einen Wertetyp, sondern um einen Referenztyp.

Der Wert eines Referenztyps ist die Referenz (Speicheradresse) auf ein Objekt des entsprechenden Typs. Beim Kopieren einer Referenz wird nicht der Wert des Objektes, auf das die Speicheradresse zeigt, kopiert, sondern nur die Referenz auf das Objekt selbst. Der Datentyp string wird dabei intern selbst wie eine Klasse behandelt. Dieser muss jedoch nicht durch ein Modul explizit eingebunden werden, wie dies z. B. für Bibliotheksklassen der Fall ist (siehe dazu auch die Abschnitt 5.12 und 5.19).

Die Unterscheidung in Werte- und Referenztypen innerhalb der IFC-PL wird in ähnlicher Weise auch in den Programmiersprachen Java und C# gemacht und ist durch diese Vorbilder inspiriert. Der Ganzzahldatentyp Integer (int) wird in der Programmiersprache Java und C# beispielsweise jeweils als Wertetyp betrachtet. D. h. eine Variable vom Typ Integer speichert nicht einen Verweis (Adresse) auf ein Integer-Objekt, sondern den Wert der Ganzzahl selbst. Bei einem string-Datentyp wird hingegen von beiden Sprachen ein Verweis (Adresse) auf ein string-Objekt gespeichert, das die eigentliche Zeichenkette beinhaltet. Auch C++ kennt Referenz- und Wertetypen. Jedoch liegt es hier in der Hand des Programmierers, wann eine Variable als Referenz- und wann diese als Wertetyp betrachtet werden soll.

Folgendes Listing zeigt exemplarisch die Verwendung der Datentypen string und double.

```
1  module PrimitiveVariableTypes;
2
3  import Core;
4
5  const int g_Answer = 42;
6
7  void main() {
8      string message = "Hello world";
9      char charValue = 'W';
10     message[6] = charValue;
11     print(message + "!"); // Ausgabe: "Hello World!"
```

```
12
13      double a = 3.0;
14      double v = a + 0.14;
15
16      print("Value: " + v); // Ausgabe: "Value: 3.14"
17      print("Value: " + g_Answer); // Ausgabe: "Value: 42"
18  }
```

Der string-Datentyp weist einige Besonderheiten auf. Mithilfe des Indexoperators (eckige Klammern in Zeile 10) wird der Buchstabe an der Indexposition 6 verändert. Weiter kann ein string-Objekt mit anderen string-Objekten oder primitiven Datentypen zu einem einzelnen string-Objekt verkettet werden.

Globale Variablen, also Variablen, die außerhalb einer Methode, Funktion oder Klasse auftreten, werden ebenfalls unterstützt, wie dies in Zeile 5 und 17 des Listings zu sehen ist.

Für die Variablendeklaration stehen u. a. folgende syntaktische Regeln zur Verfügung (eine vollständige Auflistung aller Produktionsregeln findet sich im Anhang B.2):

⟨*var_decl*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩
| ⟨*ident*⟩ ⟨*ident*⟩ '=' ⟨*expr*⟩
| 'const' ⟨*ident*⟩ ⟨*ident*⟩ '=' ⟨*expr*⟩

Mit dem const-Modifizierer kann eine Variable als konstant definiert werden. Das heißt, der Wert einer Variablen ist nach ihrer Zuweisung unveränderlich. Der const-Modifizierer kann nur auf Wertetypen angewendet werden. Einer Variablen muss bei der Deklaration nicht zwangsläufig ein Wert zugewiesen werden, außer wenn es sich um eine Konstante handelt. Dies ermöglicht beispielsweise, mathematische Konstanten wie die Kreiszahl π oder die Eulersche Zahl e zu definieren, und stellt dabei sicher, dass die definierte Konstante nicht mit einem falschen Wert überschrieben werden kann. Zudem bieten Konstanten weitere Vorteile wie die Möglichkeit einer verkürzten Schreibweise (π ist kürzer als 3.14159265358979323846264338327950288).

Für die arithmetischen Datentypen werden implizite Typkonversionen durchgeführt. Diese verhalten sich wie bei C++.

5.8 Kontrollstrukturen

IFC-PL-Programme setzen sich aus Anweisungen (Statements) zusammen. Diese werden in ihrer angegebenen Reihenfolge nacheinander abgearbeitet (vgl. Abschnitt 5.2). Um Abzweigungen in diesem linearen Kontrollfluss zu erlauben, besitzt IFC-PL verschiedene Kontrollstrukturen.

Bedingte Anweisungen ermöglichen es, abhängig von einer Bedingung eine bestimmte Anweisung auszuführen. Dabei gliedert sich eine bedingte Anweisung in

einen Bedingungs-, einen Dann- und einen Sonst-Teil. Eine bedingte Anweisung wird durch das Schlüsselwort if eingeleitet. Daher spricht man auch von einer sogenannten if-Anweisung. Der Dann-Teil wird ausgeführt, wenn die Bedingung erfüllt ist. Entsprechend wird der Sonst-Teil ausgeführt, falls die Bedingung nicht erfüllt ist. Der Sonst-Teil wird durch das Schlüsselwort else eingeleitet. Dieser ist optional und kann auch weggelassen werden. Die Bedingung einer if-Anweisung gilt als erfüllt, wenn diese einen Wert ungleich dem boolschen Wert false besitzt. Dabei wird auch der Wert 0 bzw. null (siehe Abschnitt 5.12) als false interpretiert. Damit ermöglicht eine if-Anweisung die Wahl zwischen zwei verschiedenen Ausführungspfaden.

⟨*if_stmt*⟩ ::= 'if' '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩ 'else' ⟨*stmt*⟩
| 'if' '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩

Eine konkrete Anwendung der zweiten Regel zeigt folgendes Beispiel:

```
1 if (a > b) {
2     angle *= -1;
3 }
```

Eine andere Möglichkeit der Verzweigung stellt die switch-Anweisung dar:

⟨*switch_label*⟩ ::= 'case' ⟨*expr*⟩ ':'
| 'default' ':'

⟨*switch_section*⟩ ::= ⟨*switch_label*⟩ ⟨*stmts*⟩

⟨*switch_sections*⟩ ::= ⟨*switch_section*⟩
| ⟨*switch_sections*⟩ ⟨*switch_section*⟩

⟨*switch_stmt*⟩ ::= 'switch' '(' ⟨*expr*⟩ ')' '{' '}'
| 'switch' '(' ⟨*expr*⟩ ')' '{' ⟨*switch_sections*⟩ '}'

Die switch-Anweisung wertet einen gegebenen Ausdruck aus und springt dann zur entsprechenden Fallbehandlung (case-Anweisung). Eine case-Anweisung darf nur konstante Ausdrücke enthalten. Sollte der entsprechende Vergleichsausdruck mit keinem der case-Anweisungen übereinstimmen, so wird in den default-Abschnitt gesprungen, insofern dieser vorhanden ist. Ist kein default-Abschnitt vorhanden und keine entsprechende Fallbehandlung vorgesehen, so wird an das Ende der switch-Anweisungen gesprungen. Das Switch-Statement verhält sich ähnlich wie das C++-Switch-Statement.

IFC-PL unterstützt verschiedene Schleifenarten: for-, while- und do-while-Schleifen. Diese Schleifenarten wurden aus der Programmiersprache C übernommen und haben ihren Einzug auch in die Programmiersprachen wie C++, Java und C# gefunden.

Die allgemeine Notation einer for-Schleife ist im Folgenden zu sehen:

⟨*for_stmt*⟩ ::= 'for' '(' ⟨*for_init_stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*for_iterator*⟩ ')' ⟨*stmt*⟩
| 'for' '(' ⟨*for_init_stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*for_iterator*⟩ ')' '{' '}'

Eine for-Schleife, die fünfmal iteriert, wird im folgenden Listing gezeigt:

```
1  const int iterations = 5;
2
3  for (int i = 1; i < iterations+1; i++) {
4      double sign = i % 2 == 0 ? 1 : -1;
5      // ...
6  }
```

Neben for-Schleifen werden auch while- und do-while-Schleifen unterstützt. Die Syntax zu diesen Schleifen ist im Folgenden abgebildet:

⟨*while_stmt*⟩ ::= 'while' '(' ⟨*expr*⟩ ')' ⟨*stmts*⟩
| 'while' '(' ⟨*expr*⟩ ')' '{' '}'

⟨*do_while_stmt*⟩ ::= 'do' '{' '}' 'while' '(' ⟨*expr*⟩ ')'
| 'do' '{' ⟨*stmts*⟩ '}' 'while' '(' ⟨*expr*⟩ ')'

Eine while-Schleife wird wiederholt, solange ihre Ausführungsbedingung erfüllt ist. Ist die Ausführungsbedingung von Anfang an nicht erfüllt, so wird die Schleife kein einziges Mal durchlaufen. Im Gegensatz dazu wird bei einer do-while-Schleife der Schleifenkörper zunächst durchlaufen und erst dann die Schleifenbedingung überprüft. Ist die Bedingung erfüllt, wird der Schleifenkörper der do-while-Schleife ein weiteres Mal durchlaufen. Dieser Vorgang wiederholt sich, solange die Schleifenbedingung erfüllt ist. Der Schleifenkörper einer do-while-Schleife wird also immer mindesten einmal durchlaufen, unabhängig von der Schleifenbedingung.

5.9 Funktionen

IFC-PL erlaubt im Unterschied zu C# bzw. Java die Definition von Funktionen, was im folgenden Beispiel demonstriert wird:

```
1  module Factorial;
2
3  import Core;
4
5  int factorial(const int n) {
6      if(n==1)
7          return 1;
8      else
9          return n * factorial(n-1);
10 }
11
12 void main() {
13     print(factorial(5)); // Output: "120"
14 }
```

Das gezeigte Programm ruft in der main-Funktion die definierte factorial-Funktion auf. Die Syntax für Funktionsdefinitionen lautet:

⟨*func_decl*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*ident*⟩ ⟨*ident*⟩ '+' '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩

Die main-Funktion hat eine Sonderstellung. Diese dient als Einstiegspunkt für ein IFC-PL-Programm. Eine main-Funktion ist aber nicht zwingend nötig (siehe Abschnitt 5.3).

Funktionen sind für alle Code-Fragmente im gleichen Modul sichtbar. Die Reihenfolge der Definition von Funktionen spielt dabei keine Rolle. Ebenfalls sind Funktionen in Modulen sichtbar, die das Modul importieren, indem die Funktion definiert wurde. Funktionen können überladen werden, das heißt, sie können mehrfach mit unterschiedlichen Parameterlisten definiert werden. Beim Aufruf einer überladenen Funktion wird diejenige gewählt, die mit der minimalen Anzahl an impliziten Typkonversionen in Frage kommt.

Der const-Modifizierer kann auch bei Übergabeparametern in Funktionen verwendet werden. Hierbei müssen die konstanten Werte erst beim Aufruf einer Funktion festgelegt werden. Der const-Modifizierer ist dabei auf Variablen beschränkt, die Wertetypen definieren, und kann nicht auf Referenztypen angewendet werden.

Als Implementierer einer Funktion betrachtet man Funktionsparameter oft als Eingabeparameter, die von der Außenwelt an die Funktion überreicht werden. Dabei wird oft implizit davon ausgegangen, dass ein Funktionsparameter den Wert enthält, der von der Außenwelt stammt. Jedoch ist es möglich, einen nichtkonstanten Parameter innerhalb einer Funktion mit einem neuen Wert zu überschreiben, der nicht von der Außenwelt stammt, der also von dem Wert, mit dem der Funktionsparameter beim Funktionsaufruf initialisiert wurde, abweicht. Geht ein Entwickler bei einer späteren Abänderung einer Funktion davon aus, dass die Funktionsparameter die nichtmodifizierten Werte der Außenwelt enthalten, diese jedoch tatsächlich innerhalb der Funktion (ohne Kenntnis des Entwicklers darüber) durch neue Werte überschrieben wurden, kann es zu falschen Annahmen über den Programmcode kommen, der zu Softwarefehlern führen kann. Mit dem const-Modifizierer kann ein Programmierer zusichern, dass ein Funktionsparameter den Wert der Außenwelt enthält und nicht einen modifizierten Wert, und so helfen, derartige Softwarefehler zu vermeiden. Genau aus diesem Grund wurde dieses Konzept aus der Programmiersprache C bzw. C++ in die IFC-PL übernommen. In dieser Form existiert dieses Konzept nicht in den Programmiersprachen Java oder C#.

5.10 Enumerationen

Ein Aufzählungstyp wird durch das Schlüsselwort enum definiert. Dabei kann man dem Enumerationstyp einen beliebigen Namen zuweisen und im Anschluss im Enumerationskörper (enum_body) die verschiedenen Attribute auflisten:

⟨*enum_body*⟩ ::= ⟨*ident*⟩
| ⟨*enum_body*⟩ ',' ⟨*ident*⟩

⟨*enum_decl*⟩ ::= 'enum' ⟨*ident*⟩ '{' ⟨*enum_body*⟩ '}'

Folgendes Beispiel zeigt den Umgang mit Enumerationen in der IFC-PL.

```
1  module TrafficLight;
2
3  import Core;
4
5  enum eTrafficLightState {
6      Green,
7      Yellow,
8      YellowRed,
9      Red
10 }
11
12 void main() {
13     eTrafficLightState state = eTrafficLightState.Yellow;
14
15     switch(state) {
16         case eTrafficLightState.Green:
17             print("Grün!\n");
18             break;
19         case eTrafficLightState.Yellow:
20             print("Gelb!\n");
21             break;
22             // ...
23         default:
24             print("Unbekannt!\n");
25             break;
26     }
27 }
```

Im Gegensatz zur Programmiersprache C++ ist am Ende einer Deklaration einer Enumeration kein Semikolon erforderlich (siehe Zeile 10). Optional kann man dieses aber trotzdem setzen. Der Zugriff auf Konstanten aus der Enumeration erfolgt mit dem Punktoperator (Zeile 16). Variablen, die als Typ einen Enumerationstyp besitzen, werden als Wertetypen betrachtet.

5.11 Felder und Speichermanagement

IFC-PL unterscheidet, wie bereits erwähnt, Daten- und Referenztypen in ähnlicher Weise wie C# oder Java. Felder (Arrays) fallen dabei in die Kategorie der Referenztypen. Diese werden in IFC-PL mithilfe der Syntax typeName[] arrayName; definiert. Dabei ist typeName ein Name eines beliebigen Datentyps (wie z. B. int) und arrayName der Name des Feldes. Der Speicher für ein Feld muss explizit reserviert werden. Dieser kann mit dem Schlüsselwort new allokiert werden. Um

zu kennzeichnen, dass noch kein Speicher reserviert wurde, wird der Wert eines Feldes initial auf den Wert null gesetzt.

```
1  module ArrayExample;
2
3  import Core;
4
5  void main() {
6      int[] numbers = new int[10]; // memory for 10 elements
7      for(int i = 0; i < numbers.count(); ++i)
8          numbers[i] = i*i;
9
10     print(numbers[4]); // Outputs: "16"
11 }
```

Die Anzahl der Elemente eines Arrays kann mithilfe der Methode count ermittelt werden, die jedes Array implizit bereitstellt.

Der Speicher, der mit dem Schlüsselwort new reserviert wurde, muss nicht explizit freigegeben werden. Referenztypen sind in IFC-PL mit einem Referenzzählungsmechanismus ausgestattet, d. h. es wird für jedes Objekt wie beispielsweise für ein Array gezählt, wie viele Referenzen darauf verweisen. Verweist keine Referenz mehr auf ein Objekt, so wird der Speicher des entsprechenden Objektes freigegeben. Dieses Prinzip wird in Java und C# mittels eines Garbage Collectors umgesetzt. Dieser verfolgt, wie viele Referenzen auf ein Objekt verweisen, und gibt den durch ein Objekt belegten Speicher zu einem geeigneten Zeitpunkt wieder frei. Der Speicher kann theoretisch ab dem Zeitpunkt freigegeben werden, ab dem ein Objekt durch keine Referenz des ausführenden Programms mehr erreichbar ist. Im Regelfall werden, um einen gewissen Overhead zu vermeiden, Speicherfreigaben durch einen Garbage Collector, wenn möglich, nicht unmittelbar durchgeführt, sondern zeitlich verzögert in größeren Speicherblöcken. In C++ liegt die Speicherverwaltung in Hand des Programmierers. Jedoch kann in C++ eine Referenzzählung mittels Intelligenter Zeiger (Smart Pointer) realisiert werden (z. B. mittels std::shared_ptr), die, sobald alle Referenzen auf den Speicher ihren Gültigkeitsbereich verlassen, den belegten Speicher wieder freigeben. IFC-PL besitzt einen Referenzzählungsmechanismus, der den Speicher sofort freigibt, sobald keine Referenz des Programmes mehr auf das entsprechende Objekt verweist. Ein komplizierter Garbage Collector entfällt hierbei.

Die main-Funktion, die als Einstiegspunkt für ein IFC-PL-Programm dient (siehe dazu Abschnitt 5.9), gibt es noch in einer weiteren Variante. In dieser erwartet die main-Funktion einen Parameter vom Typ string[]. Dieser Parameter wird zu Beginn mit den Kommandozeilenargumenten befüllt. Folgendes Beispiel zeigt ein Programm, das eine main-Funktion mit diesem Parameter verwendet.

```
1  module Echo;
2
3  import Core;
4
5  void main(string[] args) {
```

```
6     if(args.count() > 0) {
7         print(args[0]); // Outputs first command line argument
8     }
9 }
```

5.12 Klassen

Ein zentrales Konzept in der objektorientierten Programmierung ist die Klasse. Klassen werden in objektorientierten Programmiersprachen genutzt, um Kapselungen, Vererbung und Polymorphie von Softwarekomponenten zu realisieren, damit so eine bessere Wiederverwendbarkeit und Anpassungsfähigkeit von Programmcode erreicht wird (Martin, 2017). Eine Klasse kann, vereinfacht betrachtet, als eine Ansammlung von Attributen und Methoden betrachtet werden, die als Schablone für konkrete Instanzen, sogenannte Objekte, dient. Objekte sind Ausprägungen einer Klasse, welche die Attribute besitzen, die eine Klasse definiert. Man unterscheidet zwischen Klassen- und Instanzattributen. Letztere werden auch als Objektattribute bezeichnet. Jede Instanz besitzt ihren eigenen Satz an Instanzattributen, wohingegen Klassenattribute nur ein einziges Mal für jede Klasse angelegt werden. Methoden werden in Klassen- und Instanzmethoden bzw. Objektmethoden unterschieden. Klassenmethoden können unabhängig von einer Instanz aufgerufen werden, dürfen dabei jedoch nur auf Klassenattribute zugreifen. Instanzmethoden sind immer an eine konkrete Ausprägung einer Klasse geknüpft und können daher auch auf Instanzattribute (auch als Instanzvariablen bezeichnet) zugreifen. Daneben können Klassen verschiedene Konstruktoren definieren, die angeben, wie eine Instanz initialisiert werden soll. Ein Destruktor, der für eine Klasse definiert, wie diese wieder freigegeben wird, existiert im Sprachumfang der IFC-PL nicht, da der von IFC-PL-Instanzen belegte Speicher wieder implizit freigegeben wird.

Durch Kapselung ist es möglich, Attribute und Methoden einer Klasse von der Außenwelt abzuschirmen, diese also so zu gestalten, dass diese von außerhalb der Klasse nicht aufrufbar bzw. modifizierbar sind. Dadurch steht einem Benutzer nur eine Teilmenge der Attribute und Methoden dieser Klasse zur Verfügung, die sogenannte öffentliche Schnittstelle der Klasse. Die Interna der Klasse selbst können sich dabei zu einem späteren Zeitpunkt beliebig ändern, ohne dass diese Veränderungen einen Einfluss auf andere Softwarekomponenten, welche die öffentliche Schnittstelle dieser Klasse nutzen, nach sich ziehen. Vorausgesetzt hierbei wird natürlich, dass sich das grundlegende Verhalten der Klasse nicht verändert, diese also weiterhin den vereinbarten Dienst nach außen hin über ihre öffentliche Schnittstelle zur Verfügung stellt und sich nur interne Details, die für die Außenwelt nicht von Bedeutung sind, verändern. Kapselung bietet neben diesem, auch als Information Hiding bezeichnetem Prinzip, auch noch weitere Vorteile. Beispielsweise werden dadurch alle Attribute und Methoden einer Klasse an einem zentralen Ort gebündelt und liegen nicht verstreut über verschiedene Softwarekomponenten verteilt.

Mittels Vererbung können Attribute und Verhaltensweisen (gemeint sind Methoden) einer Klasse an eine andere Klasse (die abgeleitete Klasse) weitervererbt werden. Dadurch kann Code-Duplizierung vermieden werden, da ähnliche Objekte den gleichen Programmcode teilen können. Vererbung befähigt zur Wiederverwendbarkeit von Code, da eine abgeleitete Klasse (Unterklasse) die Methoden und Attribute der Oberklasse (Klasse, von der geerbt wird) wiederverwenden kann.

Durch Polymorphie können unterschiedliche Unterklassen auf verschiedene Weisen auf einen Methodenaufruf reagieren. Dadurch können Gemeinsamkeiten zwischen den Klassen mit dem gleichen Programmcode abgedeckt (keine Codeduplizierung) und Unterschiede zwischen Unterklassen individuell behandelt werden. Dadurch vereinfacht sich dank Polymorphie der gesamte Programmcode, da viele Fallunterscheidungen und -behandlungen für unterschiedliche Objekttypen entfallen können.

Kapselung, Vererbung und Polymorphie bilden zusammen die Grundsäulen des objektorientierten Paradigmas und besitzen noch eine Reihe weiterer Vorteile und grundlegender Überlegungen, die hinter diesen Konzepten stecken, zu denen zahlreiche Publikationen existieren wie z. B. (Scott, 2005), (Mitchell & Apt, 2001) oder (Pierce, 2002).

IFC-PL unterstützt die sogenannte Einfachvererbung, das heißt, eine Klasse kann maximal von genau einer anderen Klasse erben. Es ist im Gegensatz zur Programmiersprache C++ nicht möglich, von mehreren Klassen geleichzeitig zu erben. In C# und Java wurde diese Mehrfachvererbung, die aus C++ bekannt ist, ebenfalls stark eingeschränkt, um typische Probleme wie z. B. das Diamond-Problem (Martin, 1998) zu vermeiden. Um der zusätzlichen Sprachkomplexität zu entgehen, die sich durch die Unterstützung eines Mehrfachvererbungskonzepts ergibt, wurde darauf in der IFC-PL verzichtet.

Die Definition einer Klasse und Instanziierung ist der Vorgehensweise in C# und Java sehr ähnlich. Durch das Schlüsselwort class, gefolgt von einem Klassennamen (ident), wird eine neue Klasse definiert. Die allgemeine Klassendefinition (class_decl) unterliegt folgenden syntaktischen Regeln:

⟨*class_body*⟩ ::= ⟨*class_member*⟩
| ⟨*class_body*⟩ ⟨*class_member*⟩

⟨*class_decl*⟩ ::= 'class' ⟨*ident*⟩ '{' '}'
| 'class' ⟨*ident*⟩ '{' ⟨*class_body*⟩ '}'
| 'class' ⟨*ident*⟩ ':' ⟨*ident*⟩ '{' '}'
| 'class' ⟨*ident*⟩ ':' ⟨*ident*⟩ '{' ⟨*class_body*⟩ '}'

⟨*class_member*⟩ ::= ⟨*meth_decl*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ':' 'base' '(' ⟨*call_args*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ 'static' ⟨*var_decl*⟩ ';'
| ⟨*access_modifier*⟩ ⟨*var_decl*⟩ ';'
| ⟨*access_modifier*⟩ ⟨*array_decl*⟩ ';'

Ein etwas umfangreicheres Beispiel, das die Verwendung von Klassen zeigt, wird im Folgenden diskutiert. Zunächst der Quellcode:

```
1  module Dog;
2
3  import Core;
4
5  class Wolf {
6     public Wolf(const int age, const string name) {
7        this.age = age;
8        this.name = name;
9     }
10
11    public string getName() {
12       return name;
13    }
14
15    public virtual string toString() {
16       return "Wolf";
17    }
18
19    protected int age;
20    protected string name;
21 }
22
23 class Dog : Wolf {
24    public Dog(const int age, const string name) : base(age, name) {
25    }
26
27    public int getTaxId() const {
28       return taxIdentificationNumber;
29    }
30
31    public string toString() override {
32       return "Dog";
33    }
34
35    private int taxIdentificationNumber;
36 }
37
38 void main() {
39    Wolf[] packOfWolves = new Wolf[2];
40
41    packOfWolves[0] = new Wolf(2, "Böser Wolf");
42    packOfWolves[1] = new Dog(1, "Daisy");
43
44    for(int i = 0; i < 2; i++) {
45       print(packOfWolves[i].getName() +
46             " is a " +
47             packOfWolves[i].toString());
48    }
49 }
```

In Zeile 5 wird die Klasse Wolf definiert, in der nächsten Zeile der Konstruktor der Klasse. Dieser muss den gleichen Namen besitzen wie die entsprechende Klasse, in der er sich befindet. Der Zugriffsmodifizierer public macht es möglich, den Konstruktor außerhalb der Klasse aufzurufen.

Mithilfe von Zugriffsmodifizierern (access_modifier) wie public, protected oder private wird die Sichtbarkeit von Attributen und Methoden festgelegt. Es gibt die Sichtbarkeiten public, protected und private, die sich ähnlich wie in den Vorbildsprachen verhalten.

Variablen können als nichtveränderlich mit dem const-Schlüsselwort festgelegt werden. Ein Beispiel hierfür findet man bei den Konstruktorenparametern in Zeile 5. Die Argumente age und name sind jeweils als unveränderlich festgelegt. Ihre Werte können nur bei der Initialisierung des Wolf-Konstruktors festgelegt und im Anschluss nicht mehr verändert werden.

Das this-Schlüsselwort (siehe Zeile 7) stellt wie in Java oder C# eine Referenz auf das eigene Objekt dar. So kann beispielsweise auf verschattete Namen wie die Membervariablen/Instanzvariablen (siehe Zeile 19 und 20) der Klasse zugegriffen werden.

In Zeile 11 befindet sich eine Membermethode/Instanzmethode mit dem Namen getName. Diese liefert als Ergebnis den Wert der Instanzvariable name. Analog liefert die Methode toString in Zeile 15 den Wert "Wolf" zurück.

Für eine Klassendeklaration stehen unterschiedliche Varianten bereit. Beispielsweise beschreibt ein Doppelpunkt nach dem ident, gefolgt von einem weiteren ident (siehe Grammatik), eine Vererbung. Ein Beispiel hierfür ist in Zeile 23 zu sehen. Die Klasse Dog erbt von der Klasse Wolf, d. h. sie erbt alle Membervariablen und -methoden der Basisklasse.

Zeile 24 zeigt den Aufruf eines Basiskonstruktors mithilfe des Schlüsselworts base. Dieses ruft den Basiskonstruktor auf. In diesem Fall ist der Basiskonstruktor der Konstruktor der Klasse Wolf. Diese Syntax und Funktionsweise dieses Schlüsselworts wurden aus der Sprache C# übernommen. Die Programmiersprache Java nutzt hier eine sehr ähnliche Umsetzung, bei der anstatt des Schlüsselworts base das Schlüsselwort super genutzt wird.

Durch das Schlüsselwort virtual in Zeile 15 wird diese Methode als überschreibbar markiert, d. h. eine abgeleitete Klasse kann die Implementierung dieser Methode verändern. Dies geschieht in Zeile 31. Um eine Methode zu überschreiben, muss diese nur den gleichen Namen und die gleiche Signatur (gleiche Rückgabe- und Übergabeparameter) besitzen wie die Methode der Basisklasse, die man überschreiben möchte. Das Schlüsselwort override ist hierbei optional. Es dient dazu, einem Übersetzer (IFC-PL Compiler/Interpreter) einen Hinweis zu geben, eine Prüfung durchzuführen, die testet, ob eine Methode mit gleichem Namen und gleicher Signatur in der Basisklasse existiert.

Würde man auch die Methode getTaxId in Zeile 27 mit dem Schlüsselwort override versehen, wäre dies ein Fehler, da keine entsprechende virtuelle Methode in der Basisklasse bereitgestellt wird. Die Methode getTaxId ist eine spezifische Erweiterung der Klasse Dog, die für die Klasse Wolf keine Rolle spielt.

In der main-Funktion wird eine Array mit Platz für zwei Objekte erzeugt (Zeile 39). Im Anschluss daran wird jeweils ein Wolf- und Dog-Objekt erzeugt. Polymorphes Verhalten zeigt sich in Zeile 47. Hier wird jeweils die toString-Methode aufgerufen. Abhängig davon, ob sich hinter der Referenz ein Wolf- oder Dog-Objekt befindet, wird entweder die toString-Methode des Wolf- oder des Dog-Objekts aufgerufen.

Das obige Programm erzeugt folgende Ausgabe:

```
1 Böser Wolf is a Wolf
2 Daisy is a Dog
```

IFC-PL unterstützt, wie anhand des Beispiels gezeigt, die Konzepte Kapselung, Vererbung und Polymorphie der objektorientierten Programmierung.

Der Default-Wert einer Instanzvariable ist zunächst null. Man kann einer Instanzvariablen auch explizit den Wert null zuweisen.

Der const-Modifizierer kann auch bei Übergabeparametern in Methoden (siehe Abschnitt 5.12) verwendet werden, analog zu Übergabeparametern bei Funktionen (siehe Abschnitt 5.9). Zudem kann eine Methode als const deklariert werden (siehe Methode getTaxId in Zeile 27), was bedeutet, dass diese Methode keine Instanzvariablen der Klasse verändern kann. Instanzvariablen können in diesem Fall nur gelesen, aber nicht geschrieben werden. Dieses Konzept wurde aus der Programmiersprache C++ übernommen und existiert in dieser Form nicht in Java oder C#. Const-Methoden helfen u. a., unbeabsichtigte Veränderungen von Instanzvariablen aufzudecken, wie beispielsweise die fälschliche Zuweisung eines Wertes an eine Instanzvariablen innerhalb der Bedingungen einer if-Anweisung, anstatt des Vergleichs der Instanzvariablen mit dem zugewiesenen Wert.

Das Schlüsselwort static legt eine Methode als Klassenmethode fest. Ähnlich verhält es sich auch bei Instanzvariablen und Klassenvariablen. Hier gibt es keine bedeutenden Unterschiede zu den C++-ähnlichen Vorbildsprachen.

5.13 Interfaces

Schnittstellen können mit dem Schlüsselwort interface definiert werden. Diese können aus einer beliebigen Anzahl von Methodendefinitionen bestehen, die eine beliebige Anzahl von Parametern besitzen und optional einen bestimmten Wert zurückgeben können. Eine Schnittstelle liefert selbst keine Implementierung, sondern nur leere Methodendefinitionen ohne Methodenkörper. Darüber hinaus kann ein Interface auch keine Membervariablen besitzen. Alle Methoden einer Schnittstelle sind automatisch öffentlich sichtbar. Deshalb gibt es für Interfaces auch

keine Zugriffsmodifizierer wie etwa public, private oder protected. Folgendes Beispiel zeigt die Anwendung von Schnittstellen in der IFC-PL:

```
1  interface Material {
2      string getName();
3  }
4
5  interface IBuildingElement {
6      IMaterial getMaterial();
7      string getName();
8  }
9
10 class Wall {
11     public string getName() {
12         return "Wall";
13     }
14
15     public IMaterial getMaterial() {
16         return return getMaterial();
17     }
18
19     private IMaterial material_;
20 }
```

Das Konzept für Interfaces in IFC-PL ist den Sprachen C# und Java entnommen. Dabei erlaubt C# neben einfachen Methoden auch Eigenschaften (Properties), Indexer und Ereignisse. Diese Sprachfeatures liegen außerhalb des Sprachumfangs der IFC-PL. Java und C# erlauben die Mehrfachvererbung bei Schnittstellen. IFC-PL unterstützt diese Art der Mehrfachvererbung bei der Verwendung von Interfaces aus rein pragmatischen Gründen (Implementierungsumfang) nicht.

5.14 Up- und Downcast

Die Umwandlung eines Basisklassentypen in einen abgeleiteten Klassentypen bezeichnet man als Downcast. Andersherum wird bei der Umwandlung eines abgeleiteten Klassentyps zu seiner Basisklasse von einem Upcast gesprochen. Der Name Upcast kommt daher, dass sich üblicherweise in einer grafischen Klassenhierarchie Basisklassen über den abgeleiteten Klassen befinden, man sich also von oben nach unten (down) durch die Klassenhierarchie bewegt. Upcasts werden in IFC-PL implizit durchgeführt. Für einen Downcast ist jedoch eine spezielle Syntax erforderlich:

⟨*cast_expr*⟩ ::= 'cast' '⟨' <*ident*⟩ '>' '(' ⟨*expr*⟩ ')'

Ein Beispiel für einen Downcast ist im folgenden Programmabschnitt dargestellt:

```
1  module Casting;
2
3  import Core;
```

```
4
5  class Wolf {}
6
7  class Dog : Wolf {
8      public int getTaxId() {
9          return 1000;
10     }
11 }
12
13 void main() {
14     Wolf wolf = new Dog();
15     Dog dog = cast<Dog>(wolf);
16
17     print(dog.getTaxId());
18 }
```

In Zeile 14 wird eine Instanz der Klasse Dog erzeugt. Diese wird anschließend der Variable wolf vom Typ Wolf zugewiesen. Dabei wird der Typ der Klasse (Dog) upgecastet zum Typ Wolf. Hierfür ist keine explizite Syntax nötig. Diese Typumwandlung von Dog nach Wolf erfolgt implizit. Eine implizite Typumwandlung wird immer durchgeführt, wenn zu einer Basisklasse umgewandelt werden soll. Anders verhält es sich, wenn man nun wieder zurück in den Untertyp umwandeln will. Hier wird eine spezielle Cast-Expression-Syntax (siehe Zeile 15) benötigt. Diese startet mit dem Schlüsselwort cast und schließt in spitzen Klammern den gewünschten Zieltyp ein. In Zeile 15 soll vom Typ Wolf zum Typ Dog gecastet werden. In runden Klammern wird der Basisausdruck angegeben, der in den Zieltyp umgewandelt werden soll. Diese Umwandlung funktioniert nur, wenn eine Vererbungsbeziehung zwischen den beteiligten Typen besteht und der Ausgangstyp tatsächlich ursprünglich dem Zieltyp entsprochen hat. Sollte dies nicht so sein, wird beim Versuch eines Casts eine Ausnahme (Exception) geworfen und der Programmablauf beendet. Casting-Operatoren finden sich in ähnlicher Weise auch in anderen Programmiersprachen. C++ stellt eine Reihe von Cast-Operatoren zur Verfügung (static_cast, dynamic_cast, const_cast und reinterpret_cast), die in ihrer Syntax sehr ähnlich dem hier vorgestellten Ansatz sind. In Programmiersprachen wie C# oder Java oder C sind die Cast-Operatoren etwas dezenter. In diesen Sprachen wird der Zieltyp einfach in runden Klammern vorangestellt (Dog dog = (Dog)wolf;). Da der unbedachte Umgang mit der Cast-Operation schnell zu Fehlern führen kann, wurde eine auffällige syntaktische Umsetzung gewählt, die vom Programmierer schnell erkannt und nicht leicht überlesen werden kann. Im Allgemeinen können häufige Cast-Operationen auf eine schlechte Codequalität hindeuten, die häufig durch den richtigen Einsatz von Polymorphie vermieden werden kann. In manchen Fällen sind jedoch explizite Cast-Operatoren hilfreich.

5.15 instanceof-Operator

Das instanceof-Schlüsselwort ist ein binärer Operator, der verwendet wird, um zu testen, ob ein Objekt (eine Instanz) ein Subtyp eines gegebenen Typs ist. Die allgemeine Syntax lautet:

⟨*instanceof_expr*⟩ ::= ⟨*expr*⟩ 'instanceof' ⟨*ident*⟩

Das Ergebnis eines instanceof-Ausdrucks ist ein binärer Wert. Folgendes Beispiel veranschaulicht die Verwendung:

```
1 if(wolf instanceof Dog) {
2     Dog dog = cast<Dog>(wolf);
3     print(dog.getTaxId());
4 }
```

Insbesondere im Zusammenhang mit einem Downcast-Operator macht die Verwendung des instanceof-Ausdrucks Sinn, um einen Fehl-Cast zu vermeiden. Mit dem instanceof-Operator kann auch geprüft werden, ob ein anderer Untertyp, der in der Vererbungshierarchie vorkommt, vorliegt:

```
1  class Mineral {}
2  class Organism {}
3  class Animal : Organism {}
4  class Dog : Animal {}
5  class Cat : Animal {}
6
7  Organism a = new Cat();
8
9  a instanceof Organism; // true
10 a instanceof Animal; // true
11 a instanceof Cat; // true
12 a instanceof Mineral; // false;
```

5.16 Wertetypen als Referenztypen

Variablen, denen eine Klasse als Datentyp zugrunde liegt, werden als Referenztypen behandelt. Gleiches gilt für Felder (Arrays). Enumerationen und einfache Datentypen wie z. B. Integer oder Char werden als Wertetypen behandelt. Diese Unterscheidung zwischen Referenz- und Wertetypen ist auch in den Sprachen Java und C# anzutreffen. Jedoch kann es in manchen Situationen erforderlich bzw. hilfreich sein, auch einen einfachen Datentyp mittels einer Referenz zu behandeln. Zu diesem Zweck wurde das Schlüsselwort ref eingeführt, das primitiven Datentypen vorangestellt werden kann, damit diese als Referenzdatentyp behandelt werden. Dieses Konzept ist in gleicher Weise in der Sprache C# anzutreffen. Folgendes Programm veranschaulicht die Verwendung dieses Schlüsselworts.

```
1 module ReferenceTypes;
2
3 import Core;
4
5 void main() {
6     int a = 5;
7     ref int b = a;
8     b = 6;
```

```
9     print(a); // Ausgabe: 6
10 }
```

Der Variablen a wurde der Wert 5 zugewiesen (Zeile 6). Im Anschluss wird eine Referenz b definiert, die sozusagen auf a verweist. Der Wert der Variablen, auf die b verweist, wird modifiziert (Zeile 8), d. h. der Wert von a wird modifiziert. Anschließend wird der Wert von a ausgegeben. Auf dem Bildschirm erscheint die Ausgabe 6.

Referenzen können auch bei der Rückgabe von Werten verwendet werden:

```
1  import Core;
2
3  class SimpleIntArray {
4     public SimpleIntArray() {}
5
6     public ref int ValueAtIndex(int index) {
7        return values[index];
8     }
9
10    private int[] values = new int[10];
11 }
12
13 void main() {
14    SimpleIntArray a = new SimpleIntArray();
15    a.ValueAtIndex(0) = '13';
16    print("Value = " + a[0] + "\n"); // Ausgabe: 13
17 }
```

Referenzen in IFC-PL sind in diesem Fall den Referenzen von C++ sehr ähnlich. In einem analogen C++-Programm würde man hier im Wesentlichen das ref-Schlüsselwort durch das & ersetzen. In C# würde man obiges Programm mithilfe von Properties realisieren. In Java müsste man den Wert in ein Transport-Referenz-Objekt verpacken und entsprechend eine Referenz auf den primitiven Datentypen nach außen weiterreichen (sogenanntes Boxing/Unboxing).

Referenztypen werden ebenfalls als Parameter in Funktionen und Methoden unterstützt, wie folgendes Listing beispielhaft zeigt:

```
1  void sumIt(const double a, const double b, ref double sum) {
2     sum = a + b;
3  }
```

5.17 Überladung von Operatoren

Die Überladung von Operatoren (wie z. B. +, -, *, /) kann helfen, Programmcode intuitiver lesbar zu machen. IFC-PL unterstützt auch dieses Konzept. Eine Addition von zwei Vektoren vereinfacht sich dadurch beispielsweise auf die Anweisung a + b.

Folgendes Programm zeigt die Überladung von Operatoren:

```
1  module OperatorOverloading;
2
3  import Core;
4
5  class Vector2i {
6     public Vector2i(const int x, const int y) {
7        values_[0] = x;
8        values_[1] = y;
9     }
10
11    public ref int operator[](const int index) {
12       return values_[index];
13    }
14
15    private int[] values_ = new int[2];
16 }
17
18 Vector2i operator+(const Vector2i a, const Vector2i b) {
19    return new Vector2i(a[0]+b[0], a[1]+b[1]);
20 }
21
22 void main() {
23    Vector2i a = new Vector2i(1,2);
24    Vector2i b = new Vector2i(2,2);
25
26    print("(" + a[0] + ", " + a[1] + ")\n"); // Ausgabe: (1, 2)
27
28    a[0] = 2;
29
30    print("(" + a[0] + ", " + a[1] + ")\n"); // Ausgabe: (2, 2)
31
32    Vector2i c = a + b;
33    print("Result (" + c[0] + ", " + c[1] + ")\n"); // Ausgabe: Result (4, 4)
34 }
```

Die Überladung von Operatoren wird ebenfalls von den Sprachen C# und C++ unterstützt. Java unterstützt diese Konzept für benutzerdefinierte Datentypen jedoch nicht.

5.18 Exceptions

Der Exception-Mechanismus ist in IFC-PL sehr eingeschränkt. Es gibt nur die Möglichkeit, eine Exception zu werfen, aber keine, um diese zu fangen. IFC-PL-Excpetions können nur von der IFC-PL-Laufzeitumgebung gefangen werden. Dies wird näher im Abschnitt 5.3 beschrieben.

Die allgemeine Syntax für Exceptions ist definiert durch:

⟨*throw_stmt*⟩ ::= 'throw' 'new' ⟨*expr*⟩

Ein einfaches Beispiel wird im Folgenden gezeigt:

```
1 module ExceptionExample;
2
3 import Exception;
4
5 void main() {
6     throw new Exception("Error.");
7 }
```

In Zeile 3 wird das Ausnahmemodul Exception importiert. Dieses gehört zur Standardbibliothek von IFC-PL und stellt eine Klasse namens Exception bereit, deren Konstruktor einen String erwartet. Der String dient der Übergabe von Logging- bzw. Fehlernachrichten. Das Werfen einer Exception bricht das gesamte Programm sofort ab. Es darf nur die Exception von Typ Exception geworfen werden. Andere Exceptiontypen werden nicht unterstützt. Insbesondere gibt es keine Möglichkeit, eigene Exceptionklassen zu definieren.

Die allgemeine Grammatik lässt zwar beliebige Expressions zu, jedoch ist dies informal verboten. Nach dem new wären laut der Grammatik (siehe Anhang B.2) beliebige Ausdrücke (expr) wie z. B. MyClass(errorCode) möglich,. Dies wird aber nicht von der IFC-PL unterstützt. Die Grammatik könnte entsprechend modifiziert werden, um diesem Umstand Rechnung zu tragen. Die beschriebene informelle Regel muss entsprechend in einem IFC-PL-Übersetzer berücksichtig werden.

Der hier vorgestellte Ansatz für Exceptions wurde aus rein pragmatischen Gründen gewählt. Ein umfassenderes Exception-Konzept, das die Möglichkeiten zum Fangen und Weiterwerfen von Exceptions bietet, hätte die Realisierung der IFC-PL aufwändiger gemacht. Gleichzeitig soll aber kein Nutzer der IFC-PL gezwungen werden, ein veraltetes Konzept für Fehlerzustände wie etwa Rückgabewerte nutzen zu müssen. Der kleinste Kompromiss hierbei war folglich ein sehr eingeschränktes Exception-Konzept. Dieses ist ausreichend, um entsprechende Fehlermeldungen an die ausführende Umgebung weiterzuleiten.

5.19 Die IFC-PL-Standard-Bibliothek

Um das Schreiben von Programmen mittels der IFC-PL zu erleichtern, werden in der IFC-PL-Standard-Bibliothek Funktionen und Klassen bereitgestellt, die häufig benötigte Funktionalitäten wie Ausgabenmöglichkeiten oder mathematische Objekte bereitstellen. Die verfügbaren Funktionalitäten sind thematisch in Module untergliedert.

Das Modul Core stellt die Funktionen print, assert und exist bereit. Mit der Funktion print sind Ausgaben auf das Kommandozeilenfenster möglich (siehe Abbildung 5.4). Die Funktion assert kann genutzt werden, um Zusicherungen wie z. B. Vor- oder Nachbedingungen zu formulieren. Dabei erwartet die Funktion einen booleschen Ausdruck. Ist der Wert des booleschen Ausdrucks false, so wird eine

Ausnahme (Exception) geworfen und der Programmablauf beendet. Andernfalls läuft das Programm ohne Unterbrechung weiter. Um zu prüfen, ob eine Datei existiert, kann die Funktion exist verwendet werden. Diese erwartet den Namen einer Datei als string-Wert. Existiert die Datei, wird true zurückgegeben, andernfalls false.

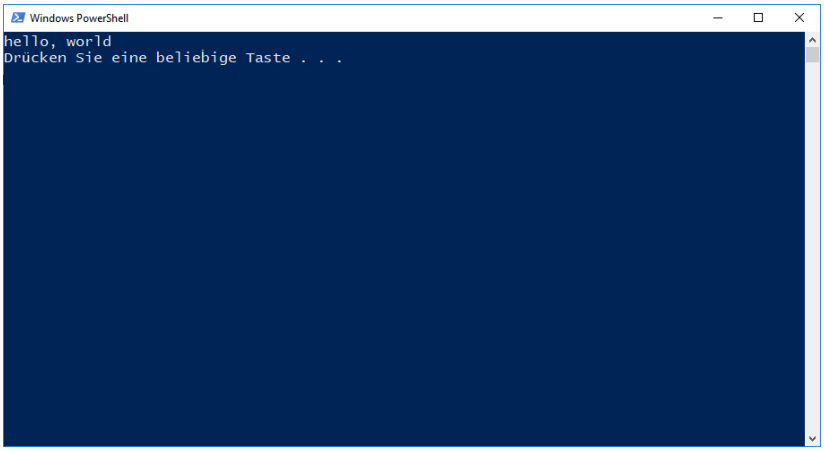


Abbildung 5.4: Die Funktion print kann für Ausgaben genutzt werden. Hier wird die Ausgabe des Hallo-Welt-Programms aus dem Abschnitt 5.4 gezeigt.

Neben dem Core-Modul gibt es auch ein Math-Modul, das häufig verwendete mathematische Funktionen bereitstellt. Dieses beinhaltet u. a. trigonometrische Funktionen (Tabelle 5.2), hyperbolische Funktionen (Tabelle 5.3), Exponentialfunktionen (Tabelle 5.4), Potenzfunktionen (Tabelle 5.5) und Rundungsfunktionen (Tabelle 5.6). Diese verhalten sich wie die entsprechenden Funktionen der C++-Standardbibliothek. Die meisten Funktionen sind mehrfach überladen. Theoretisch hätte man in fast jedem Fall ausschließlich mit dem Datentyp double arbeiten können. Dies würde aber dazu führen, dass auf Basis von float berechnete Ergebnisse plötzlich eine double-Genauigkeit besäßen, was natürlich unerwünscht ist. Daneben stellt das Modul Math noch weitere hilfreiche mathematische Funktionen bereit (siehe Tabelle 5.7), die z. B. die Umwandlung vom Bogenmaß ins Gradmaß erlauben (degreeToRadian). Zudem stehen Klassen und Funktionen für Vektor- und Matrixberechnungen bereit. Diese sind in Tabelle 5.8 aufgelistet.

Folgendes Beispiel zeigt, wie die Vektor- und Matrixklassen verwendet werden können:

```
1  module Vector4dTest;
2  import Core;
3  import Math;
4
5  void main() {
6      Vector4d a = new Vector4d(1, 0, 0, 1);
7      Vector4d b = new Vector4d(0, 1, 0, 0);
8      Vector4d c = a + b; // (1, 1, 0, 1)
9
10     Matrix44d rot = createRotationMatrixZ(degreeToRadian(180.0));
11     c = rot * c; // (-1, -1, 0, 1)
```

Funktionsname	Kurzbeschreibung
float sin(float x);	Sininus
double sin(double x);	Sininus
double sin(int x);	Sininus
float cos(float x);	Cosinus
double cos(double x);	Cosinus
double cos(int x);	Cosinus
float tan(float x);	Tangenz
double tan(double x);	Tangenz
double tan(int x);	Tangenz
float asin(float x);	Arkussinus
double asin(double x);	Arkussinus
double asin(int x);	Arkussinus
float acos(float x);	Arkuskosinus
double acos(double x);	Arkuskosinus
double acos(int x);	Arkuskosinus
float atan(float x);	Arkustangens
double atan(double x);	Arkustangens
double atan(int x);	Arkustangens
float atan2(float y, float x);	Arkustangens Quadrantenabhängig
double atan2(double y, double x);	Arkustangens Quadrantenabhängig

Tabelle 5.2: Trigonometrische Funktionen des Moduls Math

Funktionsname	Kurzbeschreibung
float sinh(float x);	Hyperbolischer Sininus
double sinh(double x);	Hyperbolischer Sininus
double sinh(int x);	Hyperbolischer Sininus
float cosh(float x);	Hyperbolischer Cosinus
double cosh(double x);	Hyperbolischer Cosinus
double cosh(int x);	Hyperbolischer Cosinus
float tanh(float x);	Hyperbolischer Tangenz
double tanh(double x);	Hyperbolischer Tangenz
double tanh(int x);	Hyperbolischer Tangenz

Tabelle 5.3: Hyperbolische Funktionen des Moduls Math

```
12
13      print(c); // prints (-1, -1, 0, 1)
14  }
```

Neben dem Core- und Math-Modul existiert auch noch das Exception-Modul. Dieses stellt die Klasse Exception bereit, die zum Werfen von Exceptions verwendet wird (siehe dazu auch Abschnitt 5.18).

Funktionsname	Kurzbeschreibung
float exp(float x);	Exponentialfunktion
double exp(double x);	Exponentialfunktion
double exp(int x);	Exponentialfunktion
float log(float x);	Natürlicher Logarithmus
double log(double x);	Natürlicher Logarithmus
double log(int x);	Natürlicher Logarithmus
float log2(float x);	Logarithmus zur Basis 2
double log2(double x);	Logarithmus zur Basis 2
double log2(int x);	Logarithmus zur Basis 2
float log10(float x);	Logarithmus zur Basis 10
double log10(double x);	Logarithmus zur Basis 10
double log10(int x);	Logarithmus zur Basis 10

Tabelle 5.4: Exponentialfunktionen des Moduls Math

Funktionsname	Kurzbeschreibung
float pow(float base, float exponent);	Potenzieren von Werten
double pow(double base, double exponent);	Potenzieren von Werten
float sqrt(float arg);	Quadratwurzel
double sqrt(double arg);	Quadratwurzel
double sqrt(int arg);	Quadratwurzel

Tabelle 5.5: Potenzfunktionen des Moduls Math

Funktionsname	Kurzbeschreibung
float round(float arg);	Rundet den Wert
double round(double arg);	Rundet den Wert
double round(int arg);	Rundet den Wert
float ceil(float arg);	Keinster Integerwert, der größer als arg ist
double ceil(double arg);	Keinster Integerwert, der größer als arg ist
double ceil(int arg);	Keinster Integerwert, der größer als arg ist
float floor(float arg);	Größter Integerwert, der kleiner als arg ist
double floor(double arg);	Größter Integerwert, der kleiner als arg ist
double floor(int arg);	Größter Integerwert, der kleiner als arg ist

Tabelle 5.6: Rundungsfunktionen des Moduls Math

Der Umfang der IFC-PL-Standard-Bibliothek ist sehr beschränkt. Es ist durchaus denkbar, den Umfang dieser Bibliothek noch zu erweitern. Im Rahmen dieser Arbeit sind jedoch die angebotenen Funktionalitäten ausreichend.

5.20 Imported Types

Die import-Anweisung kann genutzt werden, um Module einzubinden, die in IFC-PL definiert sind. Darüber hinaus kann die import-Anweisung auch verwendet wer-

Funktionsname	**Kurzbeschreibung**
int min(int a, int b);	Ermittelt den kleinsten Wert
float min(float a, float b);	Ermittelt den kleinsten Wert
double min(double a, double b);	Ermittelt den kleinsten Wert
int max(int a, int b);	Ermittelt den größten Wert
float max(float a, float b);	Ermittelt den größten Wert
double max(double a, double b);	Ermittelt den größten Wert
double factorial(int n);	Fakultät
float lerp(float a, float b, float t)	Lineare interpolation
double lerp(double a, double b, double t)	Lineare interpolation
float degreeToRadian(float degree)	Grad in Radiant
double degreeToRadian(double degree)	Grad in Radiant
double degreeToRadian(int degree)	Grad in Radiant
float radianToDegree(float rad)	Radiant in Grad
double radianToDegree(double rad)	Radiant in Grad
double radianToDegree(int rad)	Radiant in Grad

Tabelle 5.7: Funktionen, die durch das Modul Math bereitgestellt werden

Name	**Kurzbeschreibung**
Vector2{i|f|d}	2D-Vektorklasse für {int|float|double}
Vector3{i|f|d}	3D-Vektorklasse für {int|float|double}
Matrix22{i|f|d}	2x2-Matrizenklasse für {int|float|double}
Matrix33{i|f|d}	3x3-Matrizenklasse für {int|float|double}
Matrix44{i|f|d}	4x4-Matrizenklasse für {int|float|double}
{int|float|dobule} minimalComponent(Vector{2|3}{i|f|d})	Minimale Komponente eines 3D-Vektors
{int|float|dobule} distance(Vector2{i|f|d} a, Vector2{i|f|d} b)	Abstand zweichen zwei Vektoren
Matrix22{f|d} createRotationMatrix({int|float|double} angle);	Erzeugt eine 2x2-Rotationsmatrix
Matrix22{f|d} createRotationMatrix({int|float|double} angle);	Erzeugt eine 2x2-Rotationsmatrix
Matrix44{f|d} createInverseMatrix(Matrix44{f|d} m);	Erzeugt eine inverse Matrix
Matrix44{f|d} createTranslationMatrix({float|double} x, {float|double} y, {float|double} z);	Erzeugt Translationsmatrix
Matrix44{f|d} createScalingMatrix({float|double} x, {float|double} y, {float|double} z);	Erzeugt Skalierungsmatrix
Matrix44{f|d} createRotationMatrix{X|Y|Z}({int|float|double} angle);	Rotationsmatrix um {X|Y|Z}-Achse

Tabelle 5.8: Klassen und Funktionen aus dem Modul Math für Vektor- und Matrixberechnungen

den, um Datentypen eines beliebigen EXPRESS-Schemas einzubinden. Dadurch können alle Typen sowie Entitäten samt ihrer Attribute aus einem EXPRESS-Schema direkt in einem IFC-PL-Programm verwendet werden. D. h. es können neue Datentypen über ein importiertes EXPRESS-Schema bekannt gemacht werden, ohne diese explizit mittels der IFC-PL-Syntax erneut definieren zu müssen. Genau darin liegen die große Stärke und der besondere Unterschied der IFC-PL gegenüber den C++-ähnlichen Sprachen wie C, Java oder etwa C#. Durch die import-Anweisung ist eine direkte Kopplung an das EXPRESS-Schema des Bauwerksinformationsmodells möglich. Dadurch ist es auch möglich, direkt Objekte (Instanzen) auf Basis dieses Schemas zu erstellen und zu verarbeiten. Im Folgenden wird beschrieben, wie EXPRESS spezifische Typen, Enumerationen, Aggregationsdatentypen und Entitäten aus IFC-PL-Sicht verwendet werden können.

5.20.1 EXPRESS-Typen

Benutzerdefinierte EXPRESS-Typen werden in EXPRESS durch das Schlüsselwort TYPE eingeleitet. Diese sind von den bereits in die EXPRESS-Sprache integrierten einfachen Datentypen wie z. B. INTEGER oder REAL zu unterscheiden. In der IFC-PL-Welt werden selbstdefinierte EXPRESS-Typen als IFC-PL-Klassen betrachtet. Die IFC-PL-Klasse besitzt dabei den genau gleichen Namen wie der ursprünglich benutzerdefinierte EXPRESS-Typ. Zudem stellt die IFC-PL-Klasse, die den selbstdefinierten EXPRESS-Typ realisiert, zwei öffentliche Methoden mit den Namen getValue und setValue zu Verfügung. Diese lesen bzw. schreiben den Wert eines internen privaten Attributs. Das interne private Attribut hat dabei den Typ des äquivalenten IFC-PL-Datentyps, basierend auf dem ursprünglichen EXPRESS-Typ. Tabelle 5.9 definiert, wie einfache EXPRESS-Datentypen auf IFC-PL-Datentypen abgebildet werden.

EXPRES-Typ	**IFC-PL-Attributtyp**
BINARY	string
BOOLEAN	bool
INTEGER	int
LOGICAL	logical
NUMBER	double
REAL	double
STRING	string

Tabelle 5.9: Abbildung von einfachen EXPRESS-Datentypen auf IFC-PL-Datentypen

Außerdem stellt die IFC-PL-Klasse, die den selbstdefinierten EXPRESS-Typ repräsentiert, einen Konstruktor bereit, der als Argument den entsprechenden äquivalenten IFC-PL-Datentyp erwartet. Neben diesem Konstruktor wird auch ein Default-Konstruktor ohne Argumente bereitgestellt, der das interne Attribut mit einem Default-Wert belegt.

Betrachten wir nun ein Beispiel aus dem IFC 4.1-Schema. Der selbstdefinierte Typ IfcLengthMeasure wird im IFC 4.1-Schema wie folgt definiert:

```
1 TYPE IfcLengthMeasure = REAL;
2 END_TYPE;
```

Auf IFC-PL-Seite kann dieser Typ allein aufgrund der import-Anweisung, wie im nächsten Listing gezeigt wird, verwendet werden. Dabei wird angenommen, dass sich die oben gezeigte EXPRESS-Schemadefinition in der Datei IFC4X1.exp befindet.

```
1 import IFC4X1.exp; // import IFC4X1 types and entities
2
3 void main() {
4     IfcLengthMeasure length = new IfcLengthMeasure(453.5);
5     print(length.getValue()); // prints 453.5
6     length = new IfcLengthMeasure(); // length.getValue() == 0
7     length.setValue(34.4); // asign a new value
8 }
```

Der selbstdefinierte EXPRESS-Typ IfcLengthMeasure ist im EXPRESS-Schema (IFC4X1.exp) als Alias für den einfachen EXPRESS-Datentyp REAL definiert. Durch die import-Anweisung im IFC-PL-Programm (Zeile 1) wird der EXPRESS-Typ IfcLengthMeasure automatisch im IFC-PL-Programm als Klasse IfcLength-Measure mit den ensprechenden Get- und Set-Methoden (setValue, getValue) für das interne Attribut mit dem IFC-PL-Datentyp double bereitgestellt. Der EXPRESS-Typ REAL wird hier also intern auf den IFPCL-Datentyp double abgebildet. Im vorangegangen Beispiel erwartet der Konstruktor einen Wert vom IFC-PL-Datetyp double (Zeile 4). Mithifle der Methoden setValue (Zeile 7) und getValue (Zeile 5) kann der Wert des Attributs geschrieben bzw. gelesen werden. In Zeile 6 wird der Default-Konstruktor verwendet. Dieser erwartet, wie beschrieben, keine Argumente und initialisiert das interne Attribut mit seinem entsprechenden Default-Wert.

Selbstdefinierte EXPRESS-Typen können auch Where-Regeln enthalten, die bestimmte Bedingungen an die Daten stellen, wie im folgenden Beispiel gezeigt wird:

```
1 TYPE IfcInteger = INTEGER;
2 END_TYPE;
3
4 TYPE IfcPositiveInteger = IfcInteger;
5  WHERE
6     WR1 : SELF > 0;
7 END_TYPE;
```

IfcPositiveInteger ist ein Typ mit einem Where-Constraint, das nur positive Werte zulässt. Dabei stützt sich IfcPositiveInteger auf die Definition des Datentyps IfcInteger, der wiederum auf Basis des EXPRESS-Typs INTEGER definiert ist. Der EXPRESS-Typ IfcPositiveInteger wird auf die gleichnamige IFC-PL-Klasse

IfcPositiveInteger abgebildet, die ein Attribut value enthält, das vom IFC-PL-Typ int ist (entsprechend der Tabelle 5.9). Weiterhin erbt die IFC-PL-Klasse IfcPositiveInteger von der impliziten IFC-PL-Klasse IfcInteger. Eine Variable vom Typ IfcPositiveInteger kann dadurch auch überall eingesetzt werden, wo ein Ausdruck vom Typ IfcInteger erwartet wird. Die Verletzung von Where-Constraints eines EXPRESS-Typs wird implizit beim Aufruf des Konstruktors der äquivalenten IFC-PL-Klasse oder bei der Zuweisung eines Wertes an das interne Attribut value über die entsprechende Set-Methode der äquivalenten IFC-PL-Klasse überprüft. Dabei werden auch ererbte Where-Constraints berücksichtigt. Sollte eine Where-Regel verletzt sein, so wird eine Ausnahme geworfen und die Programmausführung abgebrochen.

Theoretisch hätte man eine native Unterstützung für einen Getter/Setter-Ansatz, wie er beispielsweise in C# existiert, in die IFC-PL integrieren können. Dabei wäre es dann möglich, innerhalb eines Setters beliebige Programmausführungen zu tätigen, wie beispielsweise Where-Regeln zu überprüfen. Die Entscheidung dagegen fiel zugunsten einer übersichtlicheren Grammatik und einer einfacheren Realisierung der IFC-PL-Laufzeitumgebung.

EXPRESS bietet noch weitere Möglichkeiten, um selbstdefinierte Typen exakt zu spezifizieren. Beispielsweise ist eine genauere Spezifizierung des Datentyps REAL möglich, der intern durch eine Mantisse mit einem Exponenten dargestellt wird. Die Anzahl der signifikanten Stellen der Mantisse kann optional innerhalb eines EXPRESS-Schemas mit einem Attribut spezifiziert werden. So kann z. B. festgelegt werden, dass die Mantisse nur aus vier signifikanten Stellen bestehen darf (Schenck & Wilson, 1994). Solche und ähnliche Einschränkungen werden jedoch von dem beschriebenen Ansatz, der selbstdefinierte EXPRESS-Typen automatisch auf IFC-PL-Klassen abbildet, nicht berücksichtig, da sie im Regelfall für IFC-Modelle nicht von großer Relevanz sind. Jedoch wäre es, falls es erforderlich werden sollte, denkbar, den beschriebenen Ansatz zu erweitern, um auch diese Sonderfälle unterstützen zu können.

5.20.2 EXPRESS-Enumerationen

Das EXPRESS-Schema kennt auch Enumerationstypen, wie im folgenden Listing am Beispiel der EXPRESS-Enumeration IfcActionRequestTypeEnum gezeigt wird:

```
1 TYPE IfcActionRequestTypeEnum = ENUMERATION OF
2     (EMAIL
3     ,FAX
4     ,PHONE
5     ,POST
6     ,VERBAL
7     ,USERDEFINED
8     ,NOTDEFINED);
9 END_TYPE;
```

EXPRESS-Enumerationstypen werden auf IFC-PL-Enumerationstypen abgebildet. Der Zugriff auf den Enumerationswert PHONE der Enumeration IfcActionRequestTypeEnum auf IFC-PL-Seite ist im folgenden Listing dargestellt:

```
1 IfcActionRequestTypeEnum value = IfcActionRequestTypeEnum.PHONE;
```

5.20.3 EXPRESS-Aggregationsdatentypen

EXPRESS kennt die Aggregationsdatentypen ARRAY, BAG, LIST und SET. SET und BAG besitzen keine definierte Ordnung. ARRAY und LIST dagegen definieren eine genaue Ordnung der gespeicherten Elemente. Ein SET enthält jeden Wert genau nur einmal. Ein BAG kann gleiche Werte häufiger enthalten. IFC-PL bietet eine native Unterstützung für diese EXPRESS-Cointainertypen. Folgendes Beispiel zeigt die Verwendung von Containertypen in der IFC-PL:

```
1  List<int> myList = new List<int>();
2  Set<int> mySet = new Set<int>();
3  Bag<int> myBag = new MyBag<int>();
4  Array<int> myArray = new Array<int>();
5  List<List<int>> myList2 = new List<List<int>>();
6
7  for(int i = 0; i < 10; ++i)
8     myList.add(i);
9
10 int count = my.List.count(); // count == 10
11
12 List<IfcLengthMeasure> anotherList = new List<IfcLengthMeasure>();
13 anotherList.add(new IfcLengthMeasure(3.2));
```

Die Syntax erinnert stark an Generics aus C# oder auch an Templates aus C++. Jedoch steckt dahinter kein mächtiges Metaprogrammierungskonzept, sondern lediglich eine generische Unterstützung von genau vier Containertypen. In spitzen Klammern wird jeweils der Typ der Elemente des Containers spezifiziert. Als Typ kann auch wieder ein Containtertyp spezifiziert werden (siehe Zeile 5). Alle vier Cointainertypen (Array, Bag, List und Set) stellen eine Methode count zur Verfügung, die die Anzahl der Elemente im Container ermittelt. Weiter besitzt jede Containerklasse eine Methode add, mit der neue Elemente zum Container hinzugefügt werden können. Mithilfe des Indexoperators ([]) kann auf Elemente der Aggregationsdatentypen List und Array zugegriffen werden. Zudem kann der Indexoperator auch für die Datentypen Set und Bag verwendet werden. Hierbei ist aber nicht sichergestellt, dass ein bestimmtes Element immer den gleichen Index behält.

Im EXPRESS-Schema sind selbstdefinierte Typen auf Basis von Aggratationsdatentypen erlaubt, wie anhand des folgenden Beispiels (IfcArcIndex) aus dem IFC 4.1-Schema dargestellt wird:

```
1 TYPE IfcInteger = INTEGER;
2 END_TYPE;
3
```

```
4  TYPE IfcPositiveInteger = IfcInteger;
5   WHERE
6     WR1 : SELF > 0;
7  END_TYPE;
8
9  TYPE IfcArcIndex = LIST [3:3] OF IfcPositiveInteger;
10 END_TYPE;
```

Die Verwendung des Typs IfcArcIndex auf IFC-PL-Seite wird im folgenden Listing dargestellt:

```
1 IfcArcIndex arc = new IfcArcIndex();
2 arc.getValue().add(new IfcPositiveInteger(1));
3 arc.getValue().add(new IfcPositiveInteger(2));
4 arc.getValue().add(new IfcPositiveInteger(3));
5
6 print(arc.getValue().count()); // prints "3"
```

5.20.4 EXPRESS-Entitäten

Für jede Entität des EXPRESS-Schemas wird in der IFC-PL-Welt eine Klasse erzeugt, die einen Konstruktor ohne Parameter (Defaultkonstrukor) und einen mit genau einem Parameter erzeugt. Der Konstruktor mit genau einem Parameter ist ein Konstruktor, der einen Integer-Wert erwartet. Dieser Integer-Wert gibt die eindeutige Kennung der Entität an, die die entsprechende Entität bei der Serialisierung als STEP-Datei erhält. Möchte man diese nicht selbst festlegen, so kann man den Defaultkonstruktor verwenden und die Initialisierung dieser Werte der IFC-PL-Umgebung überlassen. Auf die Attribute einer Entität kann mittels des Punktoperators (.) zugegriffen werden. Dies soll anhand eines Beispiels veranschaulicht werden. Gegeben sei folgende Entität, die Teil des IFC 4.1-Schemas ist:

```
1  ENTITY IfcBSplineCurve
2   ABSTRACT SUPERTYPE OF (ONEOF
3     (IfcBSplineCurveWithKnots))
4   SUBTYPE OF (IfcBoundedCurve);
5     Degree : IfcInteger;
6     ControlPointsList : LIST [2:?] OF IfcCartesianPoint;
7     CurveForm : IfcBSplineCurveForm;
8     ClosedCurve : IfcLogical;
9     SelfIntersect : IfcLogical;
10  DERIVE
11    UpperIndexOnControlPoints : IfcInteger :=
12                      (SIZEOF(ControlPointsList) - 1);
13    ControlPoints : ARRAY [0:UpperIndexOnControlPoints] OF
14                  IfcCartesianPoint := IfcListToArray(
15                  ControlPointsList,0,UpperIndexOnControlPoints);
16  WHERE
17    SameDim : SIZEOF(QUERY(Temp <* ControlPointsList
   Temp.Dim <> ControlPointsList[1].Dim)) = 0;END_ENTITY;
```

Folgendes IFC-PL-Programm erzeugt zunächst eine Instanz der Entität IfcBSplineCurve. Im Anschluss wird das Attribut Degree gesetzt, das den Typ IfcInteger besitzt.

```
1 import IFC4X1.exp;
2
3 void main() {
4     IfcBSplineCurve curve = new IfcBSplineCurve();
5     curve.Degree = new IfcInteger(13);
6 }
```

Dabei muss sich das EXPRESS-Schema (IFC4X1.exp) auf der Dateiebene im gleichen Ordner befinden wie das aktuelle Modul. Innerhalb des EXPRESS-Schemas wird nach wie vor EXPRESS als Sprache verwendet, um Typen und Entitäten zu definieren.

Auch die Vererbungsbeziehungen werden auf IFC-PL-Seite abgebildet. Das heißt, man könnte auf IFC-PL-Seite einer Variablen vom Typ IfcBoundedCurve den Wert einer Variablen vom Typ IfcBSplineCurve zuweisen.

Inverse-Attribute, abgeleitete Attribute (Derived Attributes) und Where-Regeln werden auf IFC-PL-Seite ignoriert. Durch entsprechende Erweiterung dieses Ansatzes könnte man allerdings auch diese Konstrukte unterstützen.

5.20.5 Lesen und Schreiben von EXPRESS-Entitäten

Zusätzlich zu den Typen und Entitäten des Schemas werden implizit durch eine import-Anweisung noch zusätzliche Klassen zum Lesen und Schreiben von STEP-P21-Dateien (entsprechend dem importierten EXPRESS-Schema) erzeugt. Das Format von STEP-P21-Dateien wird in (ISO 10303-21:2016-03, 2016) definiert. Das UML-Klassendiagramm in Abbildung 5.5 gibt einen Überblick über die zusätzlich erzeugten Klassen:

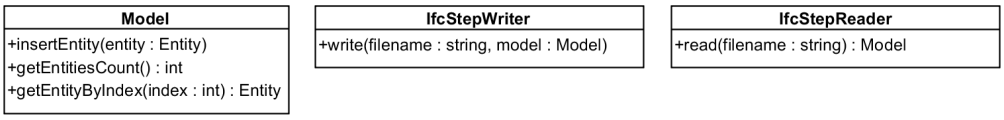


Abbildung 5.5: UML-Klassendiagramm der zuzätzlich erzeugten Klassen bei der Benutzung von Imported Types

Die Klasse `IfcStepWriter` wird genutzt, um EXPRESS-Entitäten im STEP-Format zu speichern. Analog wird die Klasse `IfcStepReader` verwendet, um STEP-Daten zu lesen und daraus entsprechende Entitäten zu verwenden. Die Klasse `Model` ist ein Container, der benutzt wird, um Entitäten zu verwalten. Die Klasse `IfcStepReader` kann aus einer IFC-Datei ein `Model`-Instanz erzeugen. Anders herum kann die Klasse `IfcStepWriter` aus einer `Model`-Instanz eine IFC-Datei erzeugen. Jede EXPRESS-Entität erbt von der impliziten IFC-PL-Klasse Entity. Dies ist notwendig, da die Model-Container-Klasse intern alle Entitäten einheitlich als Objekte der Klasse Entity betrachtet.

Folgendes Beispiel soll die Anwendung dieser Klassen zeigen. Dabei stützt sich das Beispiel auf das IFC 4.1-Schema. Ziel des Beispiels ist es, eine Entität des Typs IfcCartesianPoint zu schreiben. Die EXPRESS-Definiton der Entität IfcCartesianPoint ist im Folgenden dargestellt:

```
ENTITY IfcCartesianPoint
 SUBTYPE OF (IfcPoint);
    Coordinates : LIST [1:3] OF IfcLengthMeasure;
 DERIVE
    Dim : IfcDimensionCount := HIINDEX(Coordinates);
 WHERE
    CP2Dor3D : HIINDEX(Coordinates) >= 2;
END_ENTITY;
```

Die definierten Datentypen können im IFC-PL-Quellcode direkt verwendet werden:

```
module ImportedTypes;

import Core;
import IFC4X1.exp;

void main() {
    Model model = new Model();

    IfcLengthMeasure x = new IfcLengthMeasure(0.0);
    IfcLengthMeasure y = new IfcLengthMeasure(1.0);
    IfcLengthMeasure z = new IfcLengthMeasure(0.0);

    int id = 0;
    IfcCartesianPoint point = new IfcCartesianPoint(id++);
    point.Coordinates.add(x);
    point.Coordinates.add(y);
    point.Coordinates.add(z);
    model.insertEntity(point);

    IfcStepWriter writer = new IfcStepWriter();
    writer.write("MyFile.ifc", model);

    IfcStepReader reader = new IfsStepReader();
    Model model2 = reader.read("MyFile.ifc");
}
```

Obiges Programm erzeugt folgende Ausgabedatei:

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION(('IFC4X1'),'2;1');
FILE_NAME('filename.ifc','2017-10-27T15:44:14',(''),('',''),'','IFC4X1','
    ');
FILE_SCHEMA(('IFC4X1'));
ENDSEC;
DATA;
```

```
 8 #0=IFCCARTESIANPOINT((0.00000000000000000000,1.00000000000000000000,
       0.00000000000000000000));
 9 ENDSEC;
10 END-ISO-10303-21;
```

Das Beispiel verdeutlich auch, wie der Standalone-Betriebsmodus verwendet werden kann, um Beispieldateien für ein gegebenes EXPRESS-Schema zu erzeugen.

5.20.6 instanceof- und cast-Operator

Der instanceof- und cast-Operator können ebenfalls auf importierte Typen angewendet werden.

5.20.7 Weitere Überlegungen

Im Rahmen dieser Arbeit wurde nur die Unterstützung der Importanweisung für EXPRESS-Schemata untersucht und umgesetzt, jedoch könnte man analog bei ähnlichen Schemadefinitionen vorgehen. Beispielswiese könnte man auf ähnliche Weise XML-basierte Bauwerksinformationsmodelle unterstützen wie z. B. das ifcXML-Schema (Nisbet & Liebich, 2005) oder beispielsweise das OKSTRA-Schema (siehe Abschnitt 2.4.2).

5.21 Zusammenfassung

IFC-PL ist eine Turing-Vollständige Programmiersprache (siehe Beweis für Turing-Vollständigkeit Anhang B.4) und besitzt prozedurale und objektorientierte Sprachmerkmale, die sich sehr ähnlich zu ihren Gegenstücken in C#, Java und C++ verhalten.

Die Konzepte für Variablen, Kontrollstrukturen und Klassen wurden u. a. aus diesen Sprachen in ähnlicher Weise in IFC-PL übernommen. Hierbei wurde versucht, den kleinsten gemeinsamen Nenner zwischen diesen Sprachen zu bilden, um Programmierern, die bereits Erfahrungen mit diesen Sprachen gesammelt haben, einen schnellen Einstieg in die IFC-PL zu ermöglichen. Darüber hinaus wurden auch einzelne Sprachkonzepte übernommen, die jeweils nur in einer dieser C++-verwandten Sprachen vorkommen, z. B. Referenzen auf Wertetypen (aus C#), Funktionen (aus C++) oder der const-Modifizierer (ebenfalls aus C++). Jedoch bietet IFC-PL bei weitem nicht den Sprachumfang dieser Sprachen und besitzt im Vergleich eine weniger umfangreiche Grammatik und einige Vereinfachungen. Konzepte wie Exceptions wurden beispielsweise in einer deutlich vereinfachten Form übernommen und andere, wie z. B. Events und Delegates (aus C#), wurden gänzlich verworfen.

Durch Imported Types wird eine domänenspezifische Erweiterung für den Zugriff auf IFC-basierte Datenmodelle bereitgestellt, die den Umgang mit Bauwerksdatenmodellen, die in diesem Format gespeichert sind, erleichtert. Durch das import-

Statement können EXPRESS-Datentypen einfach eingebunden und verwendet werden.

Kapitel 6

IFC-PL-Integrationskonzept

In Kapitel 4 wurde ein programmbasierter Ansatz für die Erweiterung von Bauwerksdatenmodellen vorgestellt. Diesen Ansatz kann man auch als interfacebasierten Ansatz bezeichnen, da durch die Festlegung von Schnittstellen (Interfaces) ein flexibler Datenaustausch ermöglicht wird. Ein Vorschlag zur Implementierung dieser Schnittstellen mittels einer formalen Sprache wurde in Kapitel 5 beschrieben. In diesem Kapitel soll gezeigt werden, wie das vorgestellte Konzept auf Basis der IFC-PL realisiert und dieses in die beim Datenaustausch beteiligten Applikationen integriert werden kann. Die Umsetzung der Integration von IFC-PL in den Datenaustauschprozess erfordert folgende Arbeitsschritte:

- Definition einer Schnittstelle: Es muss eine Schnittstelle festgelegt werden. Die formale Schnittstellenbeschreibung erfolgt auf Basis der IFC-PL selbst. Die vereinbarte Schnittstelle ist als Vertragsvereinbarung des Datenaustausches zu betrachten. Diese ist sozusagen auf Schemaebene angesiedelt, jedoch nicht Bestandteil des EXPRESS-Schemas.

- EXPRESS-Schemaerstellung: IFC-PL-Schnittstellen können mittels IFC-PL-Klassen implementiert werden. Beim IFC-PL-Ansatz muss dabei auf Instanzebene in einer STEP-P21-Datei festgehalten werden, welche Instanz einer IFC-PL-Klasse für eine IFC-PL-Schnittstelle erzeugt werden und mit welchen Parametern diese initialisiert werden soll. Dies muss entsprechend bei der Erstellung bzw. bei Erweiterungen eines EXPRESS-Schemas, das solche IFC-PL-Schnittstellen unterstützen soll, berücksichtigt werden. Initialisierungsparameter müssen im IFC-PL-Ansatz in der Form eines IFC-Property-Sets bereitgestellt und ebenfalls in einer STEP-P21-Datei gespeichert werden. Dieses Property-Set wird bei der Instanziierung einer IFC-PL-Klasse an den entsprechenden Konstruktor übergeben. Damit ergibt sich an das zu erstellende EXPRESS-Schema die Mindestanforderung, dass dieses eine IfcPropertySet-Entität bereitstellen muss. Im Regelfall erweitert man in diesem Arbeitsschritt ein bestehendes IFC-EXPRESS-Schema, das diese Anforderung bereits erfüllt. Um festhalten zu können, welches IFC-

PL-Interface mit welcher IFC-PL-Klasse und welchen Initialisierungsparametern erzeugt werden soll, bietet es sich an, auf der EXPRESS-Ebene eine Entität einzuführen, die dieses Wissen vorhält. Diese Entität speichert dabei den Namen des IFC-PL-Programms, das eine spezifische IFC-PL-Schnittstelle implementiert, zusammen mit den dazugehörigen benötigten Initialisierungswerten. Dadurch kann bei Bedarf eine entsprechende Instanz erzeugt werden, die die entsprechende IFC-PL-Schnittstelle realisiert.

- Implementierung des Exports: Die Exportfunktionalität von Applikationen, die den interface-basierten Ansatz nutzen wollen, müssen angepasst werden. Auf der Seite des Exports sind neben STEP-P21-Daten auch IFC-PL-Programme (in Quelltextform) zu exportieren. Initialisierungsparameter für IFC-PL-Programme sind ebenfalls durch die Exportfunktionalität zu berücksichtigen.

- Implementierung des Imports: IFC-PL-Programme müssen von einer Applikation (Fachanwendung[1]) ausgeführt werden können, um die vereinbarten Dienste einer Schnittstellenbeschreibung verwenden zu können. Dazu muss ein Übersetzer in eine importierende Applikation implementiert werden. Der Übersetzer hat die Aufgabe, dem importierenden Programm eine Instanz eines Objektes auf Basis der Schnittstellenbeschreibung und des realisierenden IFC-PL-Programms, das als IFC-PL-Quelltext vorliegt, bereitzustellen. Da innerhalb der Schnittstellenbeschreibung auch Datentypen eines EXPRESS-Schemas vorkommen können und der Konstruktor der IFC-PL-Klasse standardmäßig immer ein IFC-Property-Set erwartet, müssen diese Datentypen auch der importierenden Applikation (Host-Applikation) bekannt gemacht werden. Unterschiedliche Applikationen sind auf Basis verschiedener Programmiersprachen implementiert. Daher muss ein Weg gefunden werden, wie zwischen der IFC-PL-Klasse und der Host-Applikation kommuniziert werden kann. Dazu wird im Rahmen der IFC-PL-Umsetzung ein Early-Binding-Generator verwendet. Dieser erzeugt für die Host-Applikation die nötigen Realisierungen der EXPRESS-Typen in der Host-Programmiersprache (Programmiersprache, in der die Host-Applikation programmiert wurde). Ein weiteres Programm, der Host-Language-Interface-Generator, übersetzt die Schnittstellendefinition in die Hostprogrammiersprache (Zielsprache). Zudem generiert der Host-Language-Interface Generator eine Stub-Implementierung auf Basis der Zielsprache für das vorgegebene Interface. Diese implementiert das definierte Interface, leitet jedoch die Ausführung der Methoden an die IFC-PL-Laufzeitumgebung weiter. Der Stub hat hier nur die Aufgabe, das Marshalling, also die Serialisierung der Methodenargumente und -rückgabewerte von der Hostsprache zur IFC-PL-Umgebung und wieder zu-

[1] Eine Fachanwendung ist eine Anwendung, die ein bestimmtes Fachthema der Anwendungsdomäne abbildet und speziell auf die Bedürfnisse dieses Fachthemas und der Fachdomäne zugeschnitten ist. Beispielsweise fällt in diese Kategorie Software, die speziell für die Überprüfung der Qualität und Integrität von Bauwerksmodellen entwickelt wurde, oder Software, die speziell für Modellierungsaufgaben des Tiefbaus (z. B. Trassierungsplanung) erstellt wurde.

rück, zu übernehmen. Die tatsächliche Ausführung der Methoden, die durch IFC-PL-Programme definiert ist, findet in der IFC-PL-Laufzeitumgebung statt.

Abbildung 6.1 stellt das Integrationskonzept der IFC-PL im Überblick dar.

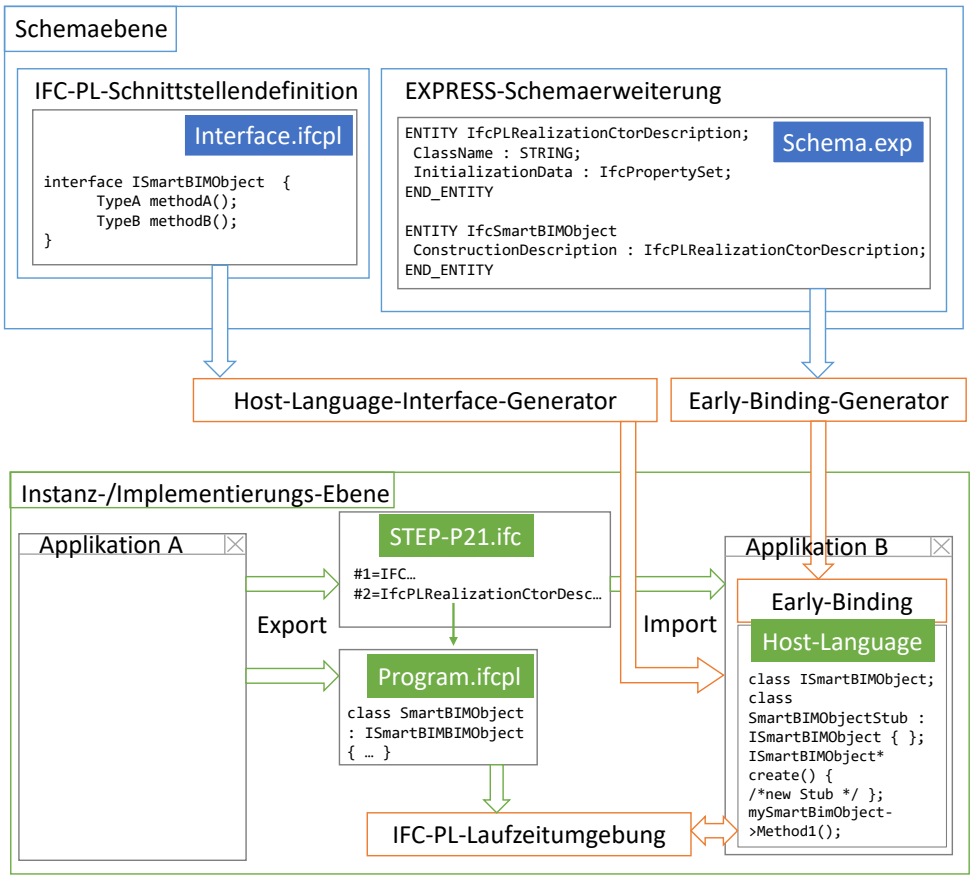


Abbildung 6.1: Integrationskonzept der IFC-PL im Überblick

Im Rahmen des Integrationskonzepts sind die folgenden Softwarekomponenten vorgesehen:

- Early-Binding-Generator: übersetzt EXPRESS-Typen in die Zielsprache der Host-Anwendung (Programmiersprache der Hostanwendung);

- Host-Language-Interface-Generator: generiert für die Zielsprache der Host-Anwendung (Fachanwendung) Programmcode, so dass diese eine vorher vereinbarte und definierte Schnittstelle nutzen kann. Dabei dient als Eingabe die Schnittstellendefinition, die mittels IFC-PL definiert wurde;

– IFC-PL-Laufzeitumgebung: übersetzt IFC-PL-Programme (IFC-PL-Quelltext) in ausführbaren Code, stellt eine Standardbibliothek für IFC-PL-Programme bereit (siehe Abschnitt 5.19), führt IFC-PL-Programme zur Laufzeit aus und behandelt Laufzeitfehler während der Ausführung eines IFC-PL-Programms.

Neben den Softwarekomponenten sind auch noch folgende Dateien notwendig:

– IFC-PL-Schnittstellendefinition: Die Schnittstelle wird formal mittels der IFC-PL beschrieben. Für Schnittstellenbeschreibungen steht das Schlüsselwort interface bereit (siehe Abschnitt 5.13).

– EXPRESS-Schema: ein EXPRESS-Schema, das die Verlinkung von IFC-PL-Programmen mit EXPRESS-Entitäten enthält.

– STEP-P21-Datei: Teil des eigentlichen Datenaustausches zwischen einer importierenden und exportierenden Applikation. Diese Datei enthält die Information, welche Entität sich auf welche IFC-PL-Programmobjekte stützt. Zudem speichert sie für jedes IFC-PL-Programmobjekt die Initialisierungswerte des Konstruktors in Form eines IFC-Property-Sets.

– IFC-PL-Programm-Datei: ebenfalls Teil des Datenaustausches. Hier wird die Realisierung einer Schnittstellendefinition mittels der IFC-PL gespeichert, d. h. der IFC-PL-Quelltext, der eine Klasse beschreibt, die die vorgegebene Schnittstelle implementiert. Alternativ hätte man den Programmcode dieser IFC-PL-Programme auch direkt in die STEP-P21-Datei mitaufnehmen können. Dies ist jedoch im derzeitigen Vorschlag nicht vorgesehen, könnte allerdings relativ einfach realisiert werden.

Im Folgenden wird noch einmal kurz eine Übersicht über die verschiedenen formalen Sprachen gegeben, die im Rahmen des Integrationskonzepts verwendet werden:

– IFC Programming Language (IFC-PL): Eigens entwickelte C++-verwandte Hochsprache (siehe Kapitel 5), die zur Definition von Schnittstellen, der Implementierung von Schnittstellen und der Entwicklung von eigenständigen Programmen (siehe Abschnitt 5.3) dient. IFC-PL-Programme werden mittels dieser Sprache beschrieben und beim Datenaustausch in Quelltextform zwischen verschiedenen Anwendungen ausgetauscht.

– Hostsprache: Die Hostsprache, auch als Zielsprache bezeichnet, ist die Programmiersprache, die von einer Anwendung für den Import von IFC-PL-Programmen verwendet wird.

– EXPRESS: Datenmodellierungssprache, auf deren Basis das IFC-Datenmodell definiert wird. Dieses wird speziell erweitert, so dass Entitäten auf IFC-PL-Programme verweisen und Initialisierungswerte für diese bereitgestellt werden können.

– STEP-P21: ISO 10303-21 (Clear Text Encoding of the Exchange Structure) wird genutzt, um Instanzdaten, die auf Basis eines EXPRESS-Schemas definiert sind, auszutauschen.

6.1 Zusammenspiel der IFC-PL-Laufzeitumgebung und der Hostapplikation

Die IFC-PL-Laufzeitumgebung stellt der Hostapplikation eine API (Application Programming Interface) bereit. Diese API ermöglicht es der Hostapplikation, IFC-PL-Programme (IFC-PL-Quelltexte) zu übersetzen. Die API besteht aus den Klassen IfcPLRuntimeEnvirionment und IfcPLRuntimeException. Die Implementierung der IFC-PL-Laufzeitumgebung muss nur einmalig durchgeführt werden und kann dann mithilfe von Language-Bindings[2] verschiedenen Programmiersprachen (Zielsprachen) als Schnittstelle angeboten werden.

Die Klasse IfcPLRuntimeEnvirionment bietet drei statische Methoden an:

– create: Die Methode create erzeugt auf Basis eines IFC-PL-Quelltextes ein Objekt in der Zielsprache, das die vereinbarte Schnittstellenbeschreibung implementiert. Dabei erwartet die create-Methode nur den Namen der Klasse, die das vereinbarte Interface implementiert, und ein IFC-Property-Set, das als Übergabeparameter an den Konstruktor der Klasse überreicht wird. Laut Konvention müssen Klassen immer in einer Datei definiert werden, die den gleichen Dateinamen besitzt (mit der Endung .ifcpl) wie die Klasse selbst. Das heißt, der Klassenname und der Dateiname (ohne Endung) müssen identisch sein.

– destroy: Die destroy-Methode löscht ein Objekt, das mit der create-Methode erstellt wurde. Dies gibt der IFC-PL-Laufzeitumgebung die Möglichkeit, reservierten Speicher oder andere reservierte Ressourcen wieder freizugeben.

– setProgramDirectory: teilt der IFC-PL-Laufzeitumgebung mit, unter welchem Pfad nach .ifcpl-Dateien gesucht werden soll. Wird über create beispielsweise eine Instanz der Klasse SmartBIMObject angefordert, so wird unter dem mit setProgramDirectory definierten Verzeichnis nach einer Datei mit dem Namen SmartBIMObject.ifcpl gesucht.

Der Host-Language-Interface-Generator stellt eine Stub-Implementierung für die Zielsprache der importierenden Applikation bereit. Beispielsweise würde für das Interface ISmartBIMObject ein Stub-Objekt mit dem Namen SmartBIMObject generiert werden. Diese Klasse bietet alle Methoden an, die in der Schnittstellenbeschreibung vereinbart worden sind. Ruft die Host-Applikation nun eine Methode des Stubs auf, so leitet diese intern den Methodenaufruf an die

[2]Ein Language-Binding ermöglicht es, von einer Programmiersprache aus Bibliotheken und Dienste zu nutzen, die nicht nativ selbst in dieser Programmiersprache entwickelt wurden.

IFC-PL-Laufzeitumgebung weiter, die dann entsprechend den IFC-PL-Quellcode ausführt und gegebenenfalls einen Rückgabewert zurückliefert. Abbildung 6.2 stellt das dynamische Zusammenspiel der Hostapplikation und der IFC-PL-Laufzeitumgebung dar.

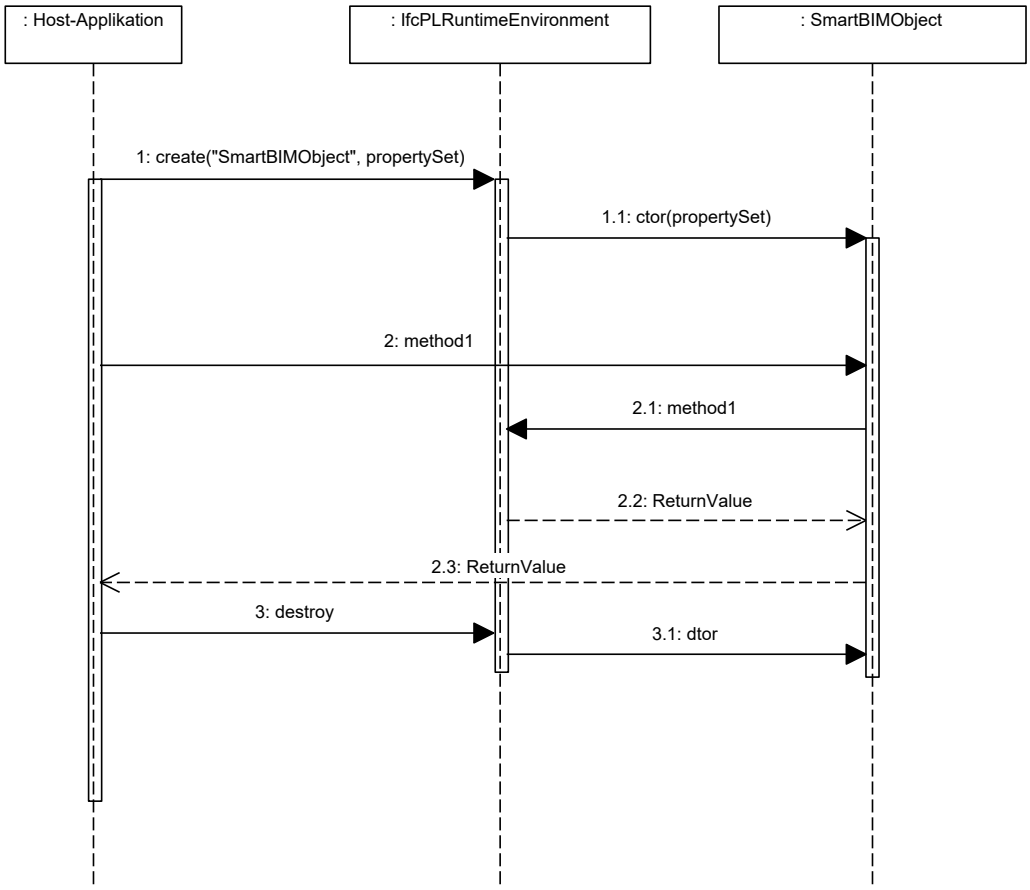


Abbildung 6.2: UML-Sequenzdiagramm, welches das dynamische Zusammenspiel der Hostapplikation und der IFC-PL-Laufzeitumgebung zeigt

Sollte es bei der Ausführung der IFC-PL-Laufzeitumgebung zu einem Problem kommen, so wird eine Exception vom Typ IfcPLRuntimeException geworfen. Alle IFC-PL-Klassen, die mit der create-Methode erstellt werden, müssen einen Konstruktor bereitstellen, der ein IfcPropertySet als Parameter erwartet. Dabei wird angenommen, dass die Schnittstellenbeschreibung ein EXPRESS-Schema importiert, das den Typ IfcPropertySet bereitstellt. Sollte der Typ IfcPropertySet nicht vorhanden sein, wird die Übersetzung eines IFC-PL-Programms von der Laufzeitumgebung mit einer Ausnahme (Exception) abgebrochen.

6.2 Beispiel: Ebene geometrische Figuren

In diesem Abschnitt sollen die vorher aufgezeigten Arbeitsschritte aus dem letzten Abschnitt anhand eines Beispiels verdeutlicht werden. Dieses Beispiel stellt keinen praktikablen Anwendungsfall dar, sondern dient lediglich der Veranschauli-

chung des Integrationskonzepts. In dem konstruierten Beispiel wird angenommen, dass ein Datenmodell erstellt werden soll, auf dessen Basis der Datenaustausch von ebenen geometrische Figuren wie Kreisen, Dreiecken, Vierecken oder Ellipsen realisiert werden soll. Um dieses Beispiel möglichst einfach zu halten, nehmen wir weiterhin an, dass im Rahmen einer Anforderungsanalyse festgestellt wurde, dass der einzige Informationsgehalt, der in diesem Datenaustauschszenario berücksichtigt werden soll, die Größe des Flächeninhalts der entsprechenden Ebenen geometrischen Figur ist.

Definition einer Schnittstelle

Aus diesen Ausgangsbedingungen kann nun eine formale Schnittstelle erarbeitet werden, die den Austausch der Größe des Flächeninhalts einer ebenen geometrischen Figur unterstützt. Die Schnittstellenbeschreibung (IGeometricShape) selbst wird dabei mithilfe der IFC-PL formal erfasst. Diese besteht nur aus einer einzigen Methode, die den Flächeninhalt der geometrischen Figur als Ergebnis zurückliefert:

```
1 inteface IGeometricShape {
2     double area() const;
3 }
```

Listing 6.1: Schnittstelle für eine ebene geometrische Figur (definiert in der Datei *IGeometricShape.ifcpl*)

Die Schnittstellenbeschreibung IGeometricShape kann durch eine IFC-PL-Klasse implementiert werden. Beispielsweise kann eine Kreis-Implementierung auf Basis der IFC-PL wie folgt realisiert werden:

```
1  import IGeometricShape;
2  import IFC4X1;
3
4  class Circle : IGeometricShape {
5      public Circle(IfcPropertySet ps) {
6          for(int i = 0; i < ps.HasProperties.count(); ++i) {
7              if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
8                  IfcPropertySingleValue s = cast<IfcPropertySingleValue>(ps.HasProperties[i]);
9                  IfcIdentifier s_id = s.Name;
10
11                 readReal(s, "radius", radius_);
12             }
13         }
14     }
15
16     private void readReal(IfcPropertySingleValue ifcPSV, string name,
17                           ref double outValue) {
18         IfcIdentifier s_id = ifcPSV.Name;
19
20         if(s_id.getValue() == name) {
21             IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
22             outValue = ifcReal.getValue();
23         }
24     }
25
26     public double area() const {
27         const double pi = 3.14159265358979323846;
28         return pi * radius_ * radius_;
29     }
30
31     private double radius_ = 0.0;
32 }
```

Listing 6.2: Kreis-Implementierung (*Circle.ifcpl*)

Die Kreis-Implementierung importiert zwei Module: das Modul IGeometricShape und das Modul IFC4X1. Im Modul IGeometricShape befindet sich die Definiton der

gleichnamigen Schnittstelle IGeometricShape. Das Modul IFC4X1 importiert das IFC 4.1-EXPRESS-Schema. Dieses wird benötigt, da der Konstruktor der Kreisimplementierung (Circle) als Eingabeparameter den Typ IfcPropertySet erwartet. Das Property-Set stellt die Attributwerte, die von der Klasse Circle benötigt werden, bereit. In diesem Fall erwartet die Klasse Circle als Parameter nur den Radius des Kreises. Der ermittelte Radius aus dem Property-Set wird in der Membervariablen radius_ abgespeichert. Zum Lesen des Property-Sets stellt die Klasse Circle die Hilfsmethode readReal bereit. Der größte Teil des Codes der Klasse Circle hat nur die Aufgabe, das übergebene Property-Set auszulesen. Die eigentliche Realisierung der Schnittstelle ist in der Methode area der Klasse Circle abgebildet. In dieser wird anhand des gegebenen Radius der Flächeninhalt des Objektes berechnet und entsprechend an die aufrufende Stelle zurückgegeben. Das Programm kann auch analog für andere ebene geometrische Figuren umgesetzt werden, z. B. für ein Rechteck (siehe Listing 6.3).

```
1  import IGeometricShape;
2  import IFC4X1;
3
4  class Rectangle : IGeometricShape {
5      public Rectangle(IfcPropertySet ps) {
6          ... // read widht_ and height_ from property set
7      }
8
9
10     public double area() const {
11         return width_ * height_;
12     }
13
14     private double width_ = 0.0;
15     rivate double height_ = 0.0;
16 }
```

Listing 6.3: Rechteck-Implementierung (*Rectangle.ifcpl*)

EXPRESS-Schemaerweiterung

Damit die Schnittstellenbeschreibung in einem EXPRESS-Datenaustausch verwendet werden kann, muss diese entsprechend in einem EXPRESS-Schema berücksichtig werden. In diesem Beispiel soll das IFC 4.1-Schema so erweitert werden, dass dieses den Datenaustauch auf Basis der Schnittstelle IGeometricShape unterstützen kann. Dazu werden im IFC 4.1-Schema die zusätzlichen Entitäten IfcPLInterfaceRealizationConstructionDescription und IfcGeometricShape eingeführt, die im Folgenden mithilfe der EXPRESS-Syntax definiert werden.

```
1  ENTITY IfcPLInterfaceRealizationConstructionDescription;
2      ClassName : STRING;
3      InitializationData  : IfcPropertySet;
4  END_ENTITY;
5
6  ENTITY IfcGeometricShape
7      ConstructionDescription : IfcPLInterfaceRealizationConstructionDescription;
8  END_ENTITY;
```

Eine Instanz der Entität IfcGeometricShape teilt einem Programm, das eine IFC-Datei in diesem (um ebene geometrische Figuren) erweiterten Format importieren will, mit, wie dieses eine entsprechende IFC-PL-Klasseninstanz erzeugen kann,

das die Schnittstelle IGeometricShape implementiert. Dabei sind im Attribut ConstructionDescription der Entität IfcGeometricShape alle notwendigen Parameter hinterlegt. Dazu gehört der Name der Klasse (ClassName), die die Schnittstelle IGeometricShape realisiert, sowie das Property-Set (InitializationData), mit dem die zu erzeugende Klasseninstanz initialisiert werden soll. Der Zusammenhang, dass eine Instanz der Entität IfcGeometricShape auf eine IFC-PL-Klasseninstanz, die die Schnittstelle IGeometricShape implementiert, abgebildet wird, ist hier nur informal festgelegt und sollte im Rahmen einer Dokumentation festgehalten werden.

Implementierung des Exports

Beim Datenaustausch muss eine STEP-P21-Datei zusammen mit den entsprechenden IFC-PL-Quelltext der referenzierten IFC-PL-Klassen ausgetauscht werden. Abbildung 6.3 zeigt in einem UML-Objektdiagramm den schematischen Aufbau einer solchen STEP-P21-Datei, die auf zwei IFC-PL-Klassen referenziert. Die Referenzierung auf IFC-PL-Klassen geschieht mittels der Instanz der Klasse IfcPLInterfaceRealizationConstructionDescription. Das Attribut Name dieser Klasse enthält als Wert den Namen der IFC-PL- Klasse, welche das Interface IGeometricShape implementiert. Der Quelltext dieser Klasse wird in einer IFC-PL-Datei erwartet, die den gleichen Namen trägt wie die Klasse selbst und zusätzlich die Endung .ifcpl besitzt. In der STEP-P21-Datei selbst taucht das Interface IGeometricShape nicht explizit auf, sondern dieses wird implizit über einen Klassennamen (die für eine Klasse steht, die dieses Interface implementiert) referenziert.

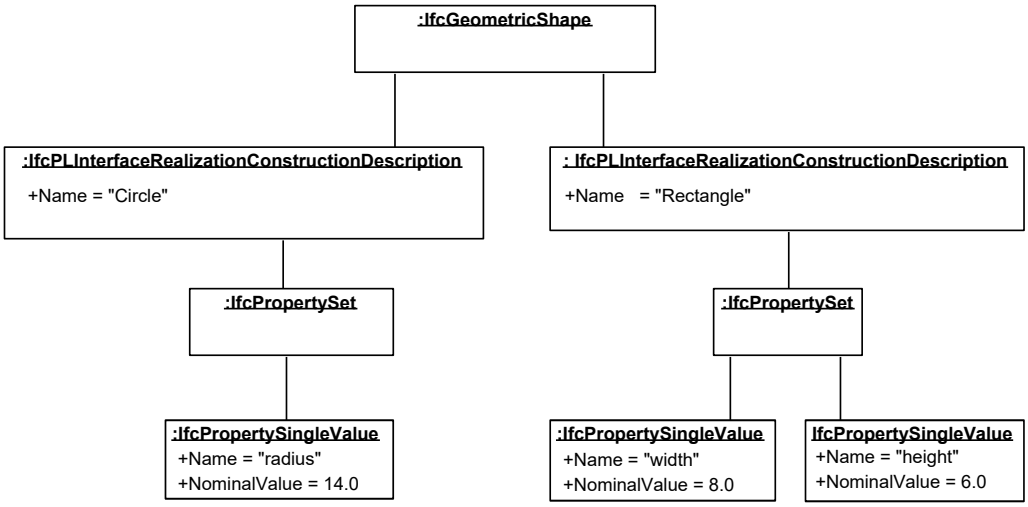


Abbildung 6.3: UML-Objektdiagramm: schematischer Aufbau einer STEP-P21-Datei, die das IGeometricShape-Interface nutzt

Implementierung des Imports

Einem Programm (Host-Programm), das die Schnittstelle IfcGeometricShape verwenden soll, muss diese Schnittstelle entsprechend bekannt gemacht werden.

Handelt es sich beim Host-Programm um ein C++-Programm, so kann mittels des Host-Language-Interface-Generators und eines Early-Binding-Generators eine entsprechende C++-Schnittstelle generiert werden, die die IFC-PL-Schnittstellenbeschreibung repräsentiert. Um eine Implementierung dieser C++-Schnittstelle zu erzeugen, kann das C++-Binding der IFC-PL-Laufzeitumgebung (IfcPLRuntimeEnvironment) verwendet werden. Diese stellt eine create-Methode bereit, mit der aus einem Property-Set und einem IFC-PL-Klassennamen eine entsprechende C++-Instanz der Schnittstelle erzeugt werden kann. Über diese Instanz kann dann auf Seite des C++-Programms der Aufruf der Methode area erfolgen, die von der erzeugten Instanz bereit gestellt wird. Intern wird der Aufruf an die IFC-PL-Laufzeitumgebung weitergeleitet, die anhand des vorliegenden IFC-PL-Quelltextes der Interface-Realisierung das entsprechende Ergebnis berechnet und dieses an die aufrufende Instanz zurückleitet.

6.3 Zusammenfassung

In diesem Kapitel wurde das IFC-PL-Integrationskonzept vorgestellt. Dabei wurde auf die unterschiedlichen Arbeitsschritte (Definition einer Schnittstelle, EXPRESS-Schemaerstellung, Implementierung des Exports, Implementierung des Imports) und die dabei entstehenden Artefakte (IFC-PL-Schnittstellendefinition, EXPRESS-Schema, STEP-P21-Datei, IFC-PL-Programm-Datei) eingegangen. Darüber hinaus wurden die verschiedenen Softwarekomponenten (Early-Binding-Generator, Host-Language-Interface-Generator, IFC-PL-Laufzeitumgebung), denen man bei diesen Arbeitsschritten begegnet, besprochen. Anhand eines Beispiels, das ebene geometrische Figuren behandelt, wurde das IFC-PL-Integrationskonzept veranschaulicht.

Kapitel 7

Prototypische Implementierung

Im Rahmen dieser Arbeit wurden verschiedene Softwareprototypen entwickelt, die in diesem Kapitel vorgestellt werden sollen.

7.1 oipExpress: Ein Early Binding Generator

Die Hauptaufgabe des TUM Open Infra Platform Early Binding EXPRESS Generators (oder kurz oipExpress) ist es, Early-Bindings[1] für C++ zu generieren. Der Quelltext von oipExpress steht unter der GNU General Public License Version 3 (Free Software Foundation, 2007) bereit[2]. Zur Unterstützung des Build-Prozesses nutzt oipExpress das Buildsystem CMake (Martin & Hoffman, 2008). Die Software gliedert sich in drei Schichten: eine Parser-, eine Metamodell- und eine Generatorschicht. Abbildung 7.1 zeigt die Bestandteile im Überblick. Die Parserschicht erwartet als Eingabe ein EXPRESS-Schema. Sollte das EXPRESS-Schema einen syntaktischen Fehler enthalten, so stoppt der Parsevorgang in der entsprechenden Zeile und gibt die korrespondierende Zeilennummer aus.

Als Ergebnis des Parsevorgangs entsteht ein Metamodell. Dieses liefert eine Beschreibung des Inhalts des EXPRESS-Schemas, auf die über verschiedene C++-Klassen, die Bestandteile dieses Metamodells sind, zugegriffen werden kann. Das Metamodell dient als Eingabe für die Generatorschicht. Diese bietet verschiedene Generatoren an, die auf Basis des Metamodells Aktionen durchführen können. Der wichtigste Generator in diesem Modul ist der C++-Early-Binding-Generator, der entsprechend für alle Entitäten und Typen des EXPRESS-Schemas C++-Klassen generiert, welche die Bestandteile eines Early-Bindings für die Programmiersprache C++ bilden. Das generierte C++-Early-Binding kann STEP-P21-Dateien lesen und schreiben. Zudem werden Dateien für das CMake-Buildsystem generiert, die

[1] Early-Bindings werden in Abschnitt 2.3.3 erläutert
[2] `http://bitbucket.org/tumcms/oipexpress`

eine einfache Einbindung der generierten Quellcodedateien als C++-Bibliothek (basierend auf diesem Buildsystem) ermöglichen.

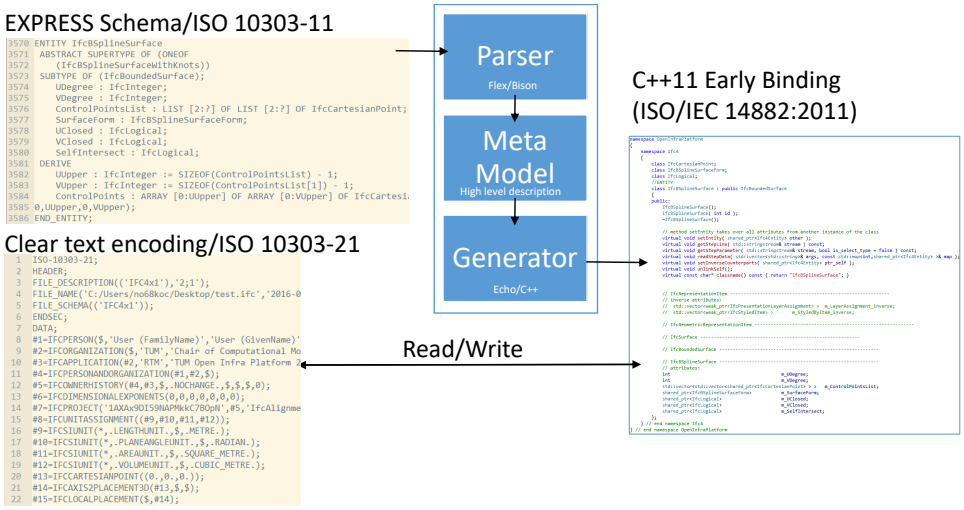


Abbildung 7.1: oipExpress im Überblick. Mittels oipExpress kann aus einem EXPRESS-Schema ein Early-Binding für die Programmiersprache C++ erzeugt werden.

Für die Umsetzung der lexikalischen Analyse nutzt oipExpress den Lexer Flex und für die syntaktische Analyse den Parser Bison (Levine, 2009). In der lexikalischen Analyse werden ca. 150 verschiedene Tokens erkannt und EXPRESS-Kommentare entfernt. Die Bison-Grammatikregeln, die von oipExpress zum Parsen des EXPRESS-Schemas verwendet werden, sind im Anhang C.1 zu finden. Dabei wurden die Regelaktionen, die beim Auffinden einer Produktionsregel durchgeführt werden, der Übersichtlichkeit halber entfernt. Der vollständige Quelltext von oipExpress samt aller Regelaktionen ist unter (CMS, 2018d) zu finden. Die exemplarische Ableitung einer Teilproduktion, basierend auf der erstellten Bison-Grammatik, ist in Abbildung 7.2 dargestellt.

oipExpress wandelt den Syntaxbaum, der durch den Parser erzeugt wird, in ein Metamodell um. Dieses stellt Klassen bereit, die Metainformationen über das EXPRESS-Schema beinhalten. Das interne Metamodell von oipExpress ist in Abbildung 7.3 dargestellt. Dieses setzt sich u. a. aus den Klassen Schema, Entity und Type zusammen. Der Parser liefert als Ergebnis nur ein Objekt vom Typ Schema. Dieses enthält eine Beschreibung aller Entitäten und Typen, die im EXPRESS-Schema enthalten sind. Mit der Methode getEntityCount bzw. getEntityByIndex der Klasse Schema können Metainformationen zu den verschiedenen Entitäten, die aus dem EXPRESS-Schema gelesen wurden, abgerufen werden. Die Klasse Entity wiederum stellt mit den Methoden getAttributeCount und getAttributeByIndex einen Mechanismus zur Verfügung, der es beispielsweise erlaubt, Informationen über die Namen und die Typen von Attributen, welche in den Entitäten enthalten sind, abzufragen. Zu EXPRESS-Typen (TYPE) können in ähnlicher Weise Metainformationen abgerufen werden.

```
TYPE IfcBoxAlignment = IfcLabel;
WHERE
   WR1 : SELF IN ['top-left', 'top-middle', 'top-right', 'middle-left',
                  'center', 'middle-right', 'bottom-left', 'bottom-middle',
                  'bottom-right'];
END_TYPE;
```

```
where_clause ->
domain_rules ->
domain_rule ->
rule_label_id  ":" expression ->
"WR1" ":" expression ->
"WR1" ":" simple_expression expression1 ->
"WR1" ":" factor expression1 ->
"WR1" ":" simple_factor expression1 ->
"WR1" ":" primary expression1 ->
"WR1" ":" qualifiable_factor expression1 ->
"WR1" ":" constant_factor expression1 ->
"WR1" ":"  built_in_constant expression1 ->
"WR1" ":"  "SELF" expression1 ->
"WR1" ":"  "SELF" rel_op_extended simple_expression ->
"WR1" ":"  "SELF" "IN" simple_expression ->
"WR1" ":"  "SELF" "IN" factor ->
"WR1" ":"  "SELF" "IN" simple_factor ->
"WR1" ":"  "SELF" "IN" aggregate_initializer ->
"WR1" ":"  "SELF" "IN" "[" aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" element "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" expression "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" simple_expression "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" term "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" factor "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" primary "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" literal "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" string_literal "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" simple_string_literal "," aggregate_initializer1 "]" ->
"WR1" ":"  "SELF" "IN" "[" 'top-left' "," aggregate_initializer1 "]" ->
...
```

Abbildung 7.2: Exemplarische Ableitung einer Produktion, basierend auf der erstellten Bison-Grammatik (siehe Anhang C.1). Die Where-Regel (gelb markiert) kann aus dem Symbol where_clause abgeleitet werden. In der Grafik ist eine Linksableitung dargestellt, d.h. es wird immer das am weitesten linksstehende Nichtterminalsymbol als nächstes ersetzt.

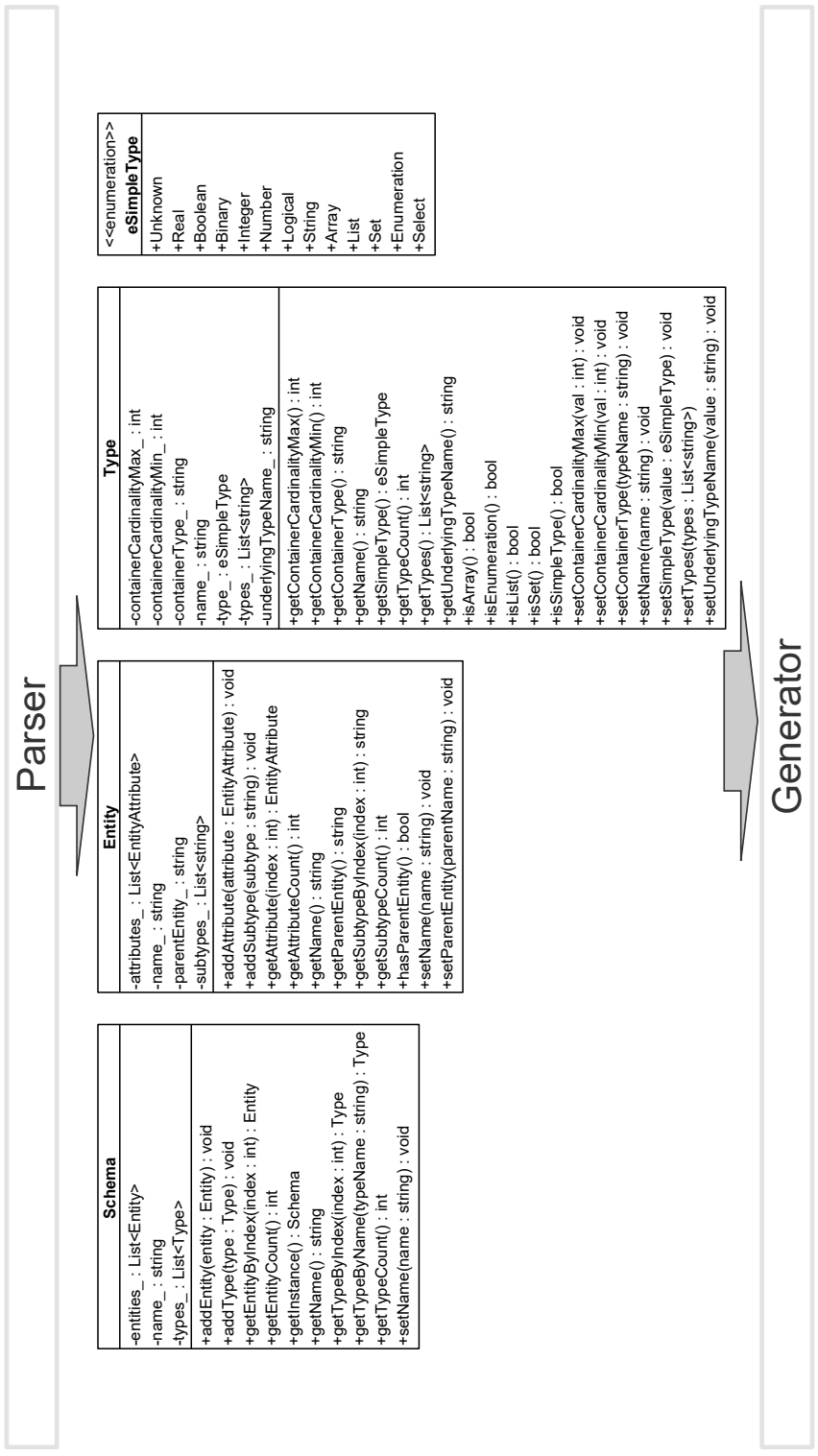


Abbildung 7.3: Überblick über die Metamodellebene von oipExpress

Die Generatorschicht bietet verschiedene Generatoren an. Jeder Generator implementiert das Generator-Interface. Dieses bietet neben einem Konstruktor und einem Destruktor nur eine Methode mit dem Namen generate an:

```
1 class Generator {
2   public:
3     Generator() {}
4     virtual ~Generator() {}
5
6     virtual void generate(std::ostream &out, Schema &schema) = 0;
7 };
```

Die generate-Methode des Generators erhält als Eingabe einen Output-Stream, in den Ausgaben, z. B. für Debugingzwecke, geschrieben werden können, und eine Instanz der Klasse Schema. Das Schema-Objekt liefert Zugriff auf das gesamte Metamodell, also auf alle Metainformationen eines EXPRESS-Schemas. Das Generator-Interface wird u. a. durch die Klassen GeneratorEcho und GeneratorOIP implementiert. Der Echogenerator hat die Aufgabe, das gelesene EXPRESS-Schema wieder unverändert auszugeben, und dient als Beispielimplementierung, die als Referenz für die Erstellung eigener Generatoren verwendet werden kann. Die Klasse GeneratorOIP generiert ein C++-Early-Binding für das entsprechende EXPRESS-Schema. Dabei werden alle Entitäten mit entsprechenden Vererbungsbeziehungen und allen Attributen als C++-Objekte abgebildet. Die generierten C++-EXPRESS-Aliasse werden im Namespace OpenInfraPlatform::<SCHEMA_NAME> eingebettet, um Namenskollisionen zu verhindern. Beispielsweise kennt IFC 2.3 und IFC 4.1 die Entität IfcDoor, die jeweils in der C++-Welt auf die Klasse IfcDoor abgebildet werden. Um Namenskollisionen zu vermeiden, wird die Entität IfcDoor aus IFC 2.3 in den Namensraum OpenInfraPlatform::IFC2X3 und die Entität IfcDoor aus IFC 4.1 in den Namensraum OpenInfraPlatform::IFC4X1 eingebettet.

oipExpress generiert mittels der Klasse GeneratorOIP für jeden EXPRESS-Typ eine C++-Klasse. Für den EXPRESS-Typ IfcBoxAlignment wird z. B. die C++-Klasse IfcBoxAlignment generiert. Da der EXPRESS-Typ IfcBoxAlignment definiert ist als TYPE IfcBoxAlignment = IfcLabel; erbt auf C++-Seite die Klasse IfcBoxAlignment von der Klasse IfcLabel. Der EXPRESS-Typ IfcLabel wird auf C++-Seite durch die Klasse IfcLabel abgebildet. IfcLabel wird im EXPRESS-Schema auf Basis des nativen EXPRESS-Datentyps STRING definiert. Dies wird in oipExpress dadurch abgebildet, dass die C++-Klasse IfcLabel ein Attribut mit dem Namen m_value und dem Datentyp std::string erhält. Ähnlich wird mit anderen nativen EXPRESS-Datentypen umgegangen. Beispielswiese wird der EXPRESS-Typ INTEGER auf den C++-Typ int abgebildet.

Die Generatorschnittstelle ermöglicht es außerdem, auf einfache Weise neue Early-Binding-Generatoren für andere Programmiersprachen zu implementieren. Beispielsweise würde der Grundstein für eine Visual Basic .NET Implementierung etwa wie folgt aussehen:

```
1 class GeneratorVBNet : public Generator {
2 public:
```

```
3       GeneratorVBNet() {
4       }
5       virtual ~GeneratorVBNet() {
6       }
7
8       void generate(std::ostream &out, OpenInfraPlatform::ExpressBinding::Schema &schema) {
9           for (int i = 0; i < schema.getEntityCount(); i++) {
10              auto &entity = schema.getEntityByIndex(i);
11
12              std::stringstream ss;
13              ss << earlyBindingDestination << "\\" << entity.getName() << ".vb";
14
15              std::ofstream ofs(ss.str(), std::ofstream::out);
16
17              ofs << "Class " << entity.getName() << std::endl;
18
19              ofs << "End Class" << std::endl;
20          }
21      }
22
23  private:
24      std::string earlyBindingDestination = "~\EarlyBindingVBNet_IFC4x1_Add1";
25  };
```

Obiges Codefragment erzeugt bereits für jede Entität des EXPRESS-Schemas eine Visual Basic .NET Klasse, die in einer eigenen Datei abgelegt wird. Mit geringem Aufwand könnten auch noch Attribute oder Vererbungsbeziehungen ergänzt werden.

Wie bereits erläutert, basiert oipExpress auf einer Schichtenarchitektur. Die Metamodellschicht kennt weder die Parserschicht noch die Generatorschicht. Die Parserschicht kennt nur die Metamodellschicht, da diese das Metamodell erzeugen muss. Diese hat jedoch keine Verbindung zur Generatorschicht. Diese wiederum kennt das Metamodell, jedoch nicht die Parserschicht.

Derzeit werden von oipExpress nur die Attribute beachtet, die später auch bei der Serialisierung in eine STEP-P21-Datei berücksichtigt werden. Derived Attributes, Where-Rules, EXPRESS-Functions oder etwa Inverse-Attribute werden vom Generator ignoriert. Diese Entscheidung wurde getroffen, da die Umsetzung dieser Funktionalitäten zum einen mehr Zeit in Anspruch genommen hätte und zum anderem dies nicht wesentlich zum Beweis der Tragfähigkeit des in dieser Arbeit beschriebenen Konzeptes beigetragen hätte. Der Parser kann allerdings einen Syntaxbaum für die genannten Sprachelemente erstellen, da die Grammatik auch diese Aspekte umfasst, jedoch werden diese aufgrund der genannten Gründe nicht im Metamodell bzw. im Generator weiterverarbeitet. Abbildung 7.4 zeigt einen Teilausschnitt des Syntaxbaumes, der für die Where-Regel aus Abbildung 7.2 generiert wird.

Wollte man auch Where-Regeln im C++-Early-Binding unterstützen, kann man z. B. auf Basis des Syntaxbaums eine Codegenerierung für C++-Code implementieren. Dabei wird jeder Knoten des Syntaxbaums besucht und entsprechend eine Codegenerierung für jeden Knoten durchgeführt. Abbildung 7.5 zeigt eine mögliche Vorgehensweise. Die C++-Klasse IfcBoxAlignment wird mit der Methode WR1 ergänzt. WR1 ist der Name der Where-Regel im IFC-EXPRESS-Schema (siehe Abbildung 7.2). Diese Methode gibt einen booleschen Wert zurück. Ist die Where-Klausel erfüllt, so wird der Wert true zurückgeliefert, andernfalls der Wert false. Der C++-Code für den Methodenkörper wird aus dem Knoten domain_rule abgeleitet. Der Knoten SELF wird entsprechend auf den C++-Code m_value

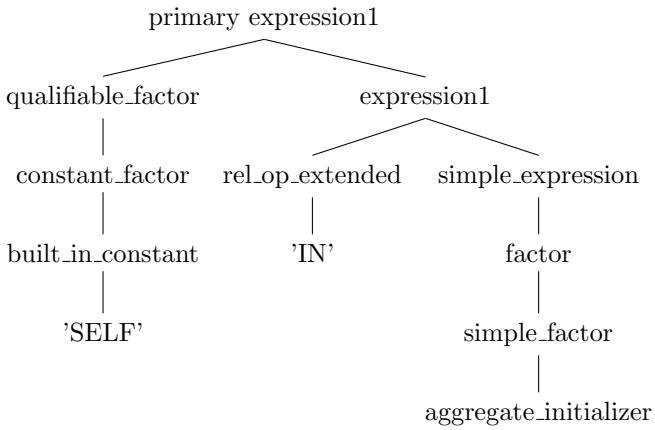


Abbildung 7.4: Teilausschnitt eines Syntaxbaumes für eine Where-Regel

abgebildet. Der aggregate_initializer wird durch einen std::vector dargestellt, der alle Werte des Aggregat-Initaliserers enthält. Als Typ für den std::vector wird der Underlying_Type verwendet. Dabei handelt es sich um den Typ, der für m_value verwendet wird. In dem gezeigten Beispiel stützt sich IfcBoxAlignment auf den EXPRESS-Typen STRING, der auf std::string abgebildet wird, d. h. als Underlying_Type wird std::string verwendet. Für den Knoten IN wird der std::vector durchsucht (mit std::find) und entsprechend true oder false, abhängig vom Suchergebnis, zurückgegeben. Ein Codegenerator müsste erkennen, dass die domain_rule eine IN-Expression enthält und dafür zunächst Code für den aggregate_initializer und danach Code zur Suche in diesem Aggregations-Initialisierer (std::find) generieren.

Der Programmcode von oipExpress ist nicht nur auf die Generierung von Early-Bindings beschränkt. Die Parserschicht kann beispielsweise ebenfalls in einem Late-Binding-Ansatz verwendet werden. Eine Late-Binding-Implementierung sollte zur Laufzeit sicherstellen, dass z. B. keine Attribute gesetzt oder Entitäten erzeugt werden, die im Schema gar nicht existieren. Dies kann ein Late-Binding nur leisten, wenn dieses das Schema kennt und über Möglichkeiten verfügt, das EXPRESS-Schema einer STEP-P21 zu reflektieren.

7.2 Die IFC-PL-Laufzeitumgebung

Der Hauptzweck der IFC-PL-Laufzeitumgebung ist es, IFC-PL-Programme auszuführen. Technisch gesehen gibt es hierfür unterschiedliche Realisierungsvarianten. Im Folgenden werden verschiedene Ansätze beschrieben. Alle Ansätze besitzen eine gemeinsame Basis: Zunächst wird in der lexikalischen Analysephase das IFC-PL-Programm in Tokens zerlegt. Im darauf folgenden Schritt wird durch einen Parser ein Syntaxbaum aufgebaut. Für allgemeine Informationen zur Realisierung von Übersetzern (Interpretern, Compilern, virtuellen Maschinen) sei hier auf die Literatur zum Compilerbau verwiesen, z. B. (Aho *et al.*, 2006), (Wirth, 2011), (Wilhelm *et al.*, 2012) oder (Appel, 1997).

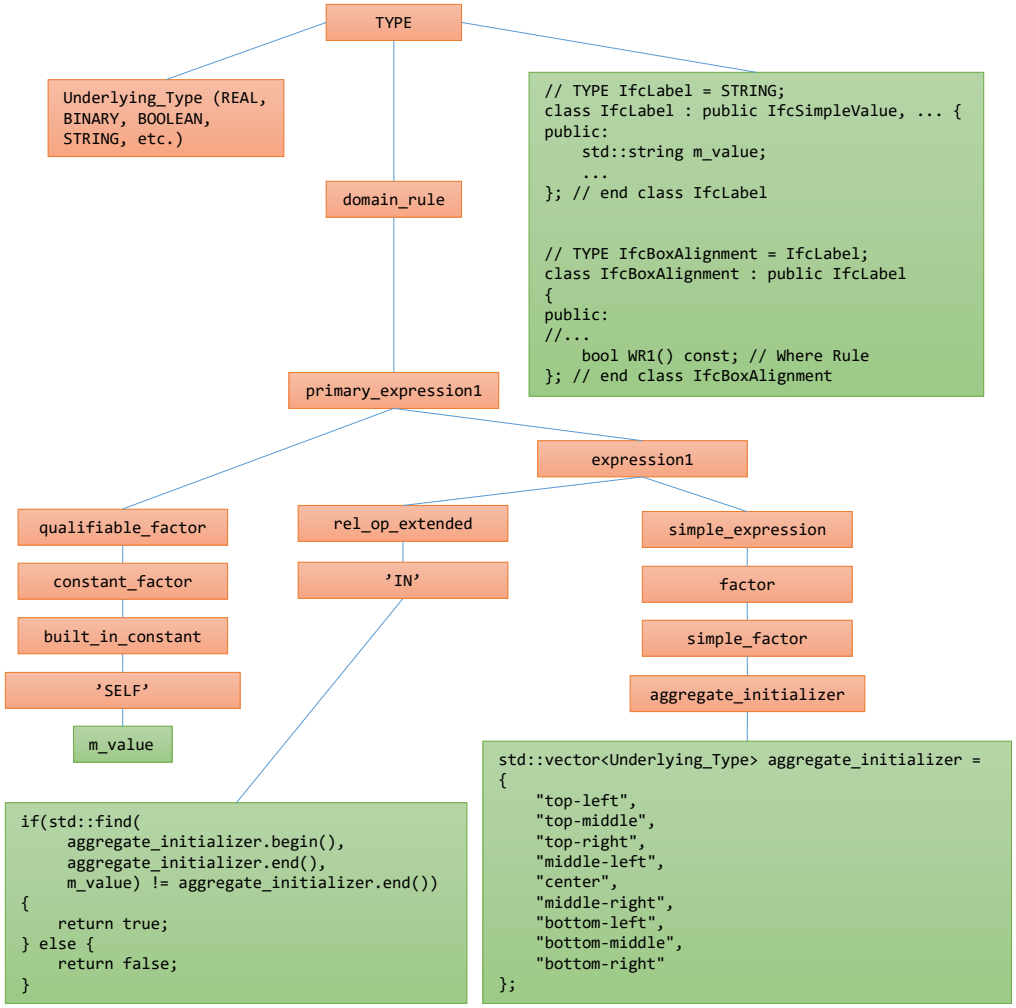


Abbildung 7.5: Mögliche C++-Codegenerierung für eine Where-Regel

Die IFC-PL wurde prototypisch mittels eines Transpilers umgesetzt. Bevor dieser Ansatz erläutert wird, sollen die folgenden Abschnitte einen Einblick in die verschiedenen Möglichkeiten der Übersetzung von IFC-PL in ausführbaren Code geben.

7.2.1 IFC-PL-Laufzeitumgebung als Interpreter

Ein Interpreter übersetzt ein Programm nicht, sondern liest, vereinfacht ausgedrückt, den Quelltext Zeile für Zeile und führt die gelesenen Anweisungen, entsprechend der Interpretation, direkt aus, ohne dabei den Quelltext in eine Maschinensprache zu übersetzen.

Wird die IFC-PL-Laufzeitumgebung als Interpreter implementiert, so kann der Syntaxbaum direkt vom Interpreter während der Laufzeit zur Ausführung eines IFC-PL-Programms genutzt werden. Im Rahmen der Entwicklung der Open Infra Platform, die im Abschnitt 7.3 gesondert beschrieben wird, wurde ein rudimentärer IFC-PL-Interpreter umgesetzt. Das Klassendesign für diesen IFC-PL-Interpreter umfasst die Klassen Token, Lexer, AbstractSyntaxTreeNode, AbstractSyntaxTree, Parser und Interpreter. Die Klasse Lexer zerlegt den Quelltext in Tokens, die dann vom Parser in einen abstrakten Syntaxbaum umgewandelt werden. Die Klasse Interpreter stellt eine Methode mit dem Namen walk bereit. Dabei besucht der Interpreter den Wurzelknoten des Syntaxbaums und der Reihe nach alle Kinderknoten und führt für die besuchten Knoten die entsprechenden interpretierten Befehle aus. Trifft der Interpreter beispielsweise auf einen Bedingungssyntaxknoten (if-Bedingung), so wird zunächst der Bedingungsausdruck ausgewertet und im Anschluss, abhängig vom Bedingungsausdruck, der Then- oder Else-Ast ausgeführt. Folgender Pseudocode, der an die Programmiersprache C++ angelehnt ist, veranschaulicht das Vorgehen des Interpreters:

```
1  class Interpreter {
2
3      void walk(AbstractSyntaxTreeNode* node) {
4          // Führe abhängig vom Knotentyp eine Aktion durch
5          switch (node->getNodeType()) {
6              ...
7              case eNodeType::Condition: {
8                      // Bedingungsausdruck berechnen
9                      walk(node->right);
10                     node->value = node->right->value;
11
12                     // Wenn Bedingung erfüllt, dann Then-Zweig ausführen:
13                     bool condition = boost::get<bool>(node->value);
14                     if(condition)
15                         walk(node->left);
16                     else
17                         walk(node->right);
18                 }
19                 break;
20     ...
```

Mit der walk-Methode werden alle Knoten des Syntaxbaums besucht. Diese besitzt im obigen Beispiel ein Switch-Statement, mit dem, abhängig vom eigentlichen Knotentyp, eine bestimmte Aktion durchführt wird. Alternativ hätte man hier auch ein Visitor Pattern realisieren können, wobei jeder Knoten mit einer visit-Methode ausgestattet worden wäre (Gamma *et al.*, 1995). Der besseren Übersichtlichkeit halber und um das Beispiel kurz zu halten, wurde der hier gezeigte Ansatz gewählt.

7.2.2 IFC-PL-Laufzeitumgebung auf Basis einer virtuellen Maschine

Anstatt den IFC-PL-Code zu interpretieren, kann dieser auch in Maschinensprache für eine konkrete CPU-Architektur, z. B. die Intel-64-Architektur (Intel Corporation, 2011), übersetzt (kompiliert) werden. Der generierte Maschinencode kann dann direkt von einer CPU ausgeführt werden. Um nicht von einer konkreten CPU-Architektur bzw. einem bestimmten Instruktionssatz einer tatsächlich existierenden CPU abzuhängen, bietet es sich an, Maschinencode für eine virtuelle Maschine zu erzeugen. Eine virtuelle Maschine ist eine idealisierte Maschine mit einem eigenen Befehlssatz. Diese existiert im Regelfall nicht real und ihre Befehle müssen erst auf einem Zielsystem implementiert werden. Ein Vorteil dieses Ansatzes ist es, dass dadurch nur die virtuelle Maschine auf verschieden Hostsysteme (Zielrechner) portiert werden und nicht der ganze Übersetzungsprozess für jede CPU-Architektur neu angepasst werden muss. Zudem besteht die Möglichkeit, den Befehlssatz der virtuellen Maschine frei wählen zu können. Dies ermöglicht es, den Befehlssatz an die zu übersetzende Programmiersprache anzupassen und dadurch den Übersetzungsprozess zu vereinfachen.

Im Rahmen dieser Arbeit wurde auf Basis der LLVM[3] Compiler Infrastructure prototypisch LLVM IR Code für eine Untermenge der IFC-PL generiert. Die LLVM Compiler Infrastructure ist eine Sammlung von modularen und wiederverwendbaren Compiler- und Toolchain-Technologien, die zur Entwicklung von Compiler-Frontends und -Backends benutzt werden können (Lattner & Adve, 2004). LLVM bietet verschiedene Module, die innerhalb eines eigenen Compilers bzw. einer eigenen Laufzeitumgebung genutzt werden können, an. Dabei nutzt LLVM eine Zwischendarstellung (Intermediate Representation), die als IR-Code bezeichnet wird. Das LLVM Framework bietet für IR-Code u. a. plattformunabhängige Optimierer, Codegeneratoren, die maschinenabhängige Assemblersprachen für eine Zielplattform erzeugen, Just-in-time Compilation (JIT) sowie einen Interpreter (virtuelle Maschine für IR-Code) an.

Die Wahl fiel auf LLVM, da virtuelle Maschinen existieren, die IR-Code ausführen können, und dieses Framework bei einer Reihe von anderen namhaften Compilern erfolgreich verwendet wird. Beispielsweise verwendet Clang (Compiler-

[3]LLVM war ursprünglich ein Akronym für *Low Level Virtual Machine*. Inzwischen ist das LLVM-Projekt jedoch von seinem Ursprung (traditionellen virtuellen Maschinen) abgerückt und hat nur noch wenig damit gemeinsam. Daher wird LLVM inzwischen nicht mehr als Akronym, sondern als Eigenname betrachtet (siehe dazu auch `http://llvm.org/`).

Frontend für die Programmiersprachen C, C++, Objective-C und Objective-C++ und Standard-Compiler für Mac OS X), Swift (Programmiersprache von Apple für iOS, macOS, tvOS, watchOS und Linux) sowie Rust (eine Systemprogrammiersprache) dieses Framework, um die Codegenerierung damit durchzuführen. Es besitzt eine BSD- bzw. MIT-ähnliche Lizenz und bietet eine gute Performance. Durch die weite Verbreitung von LLVM gibt es zahlreiche Experten, die sich mit dem Umgang dieses Frameworks auskennen, und es ist davon auszugehen, dass dieses Framework auch zukünftig noch weiterentwickelt und unterstützt wird. Technologisch gesehen hätte man hier auch die JVM (Java Virtuell Maschine) oder die CLR (Common Language Runtime) des .NET-Frameworks verwenden können, jedoch war schlussendlich auch aufgrund der Vorkenntnisse des Verfassers LLVM der gangbarere Weg.

Der Syntaxbaum, der beim Parsen eines IFC-PL-Programms entsteht, kann für die IR-Codegenerierung genutzt werden. Dabei wird jeder Knoten des Syntaxbaums besucht und entsprechender IR-Code generiert. Folgender Code zeigt einen Auszug aus der Codegenerierung mittels LLVM. Hierbei wurde für jeden Knotentyp des Syntaxbaums eine eigene Klasse eingeführt, die eine generateCode-Methode bereitstellt:

```
1  // Erzeuge Code für eine IFC-PL-Funktion
2  Value* FunctionDeclaration::generateCode(Context& context) {
3      // Erzeuge alle Funktionsparameter
4      vector<Type*> argTypes;
5      VariableList::const_iterator it;
6      for (it = arguments.begin(); it != arguments.end(); it++) {
7          argTypes.push_back(typeOf((**it).type));
8      }
9      // Erzeuge Rückgabetyp
10     FunctionType* ftype = FunctionType::get(typeOf(type), makeArrayRef(argTypes), false);
11     // Erzeuge Funktionssignatur
12     Function* function = Function::Create(ftype, GlobalValue::InternalLinkage, id.name.c_str(), context.module);
13
14     // Erzeuge Funktionskörper
15     BasicBlock* bblock = BasicBlock::Create(getGlobalContext(), "entry", function, 0);
16
17     context.pushBlock(bblock);
18
19     Function::arg_iterator argsValues = function->arg_begin();
20
21     for (it = arguments.begin(); it != arguments.end(); it++) {
22         (**it).codeGen(context);
23
24         Value* argumentValue = argsValues++;
25         argumentValue->setName((*it)->id.name.c_str());
26         StoreInst* inst = new StoreInst(argumentValue, context.locals()[(*it)->id.name], false, bblock);
27     }
28
29     // Generiere Code des Funktionskörpers
30     block.codeGen(context);
31     ReturnInst::Create(getGlobalContext(), context.getCurrentReturnValue(), bblock);
32
33     context.popBlock();
34     return function;
35 }
```

Abbildung 7.6 zeigt den LLVM-IR-Code, der für ein kleines Beispielprogramm generiert wurde. Der hier gezeigte Code ist noch nicht optimiert. LLVM kann diesen Code noch mittels verschiedener Techniken aus dem Bereich der Code Optimierung (Seidl *et al.*, 2009) optimieren.

LLVM-IR-Code kann durch eine von der LLVM Compiler Infrastructure bereitstellten Interpreter ausgeführt werden. Daneben gibt es auch Implementierungen

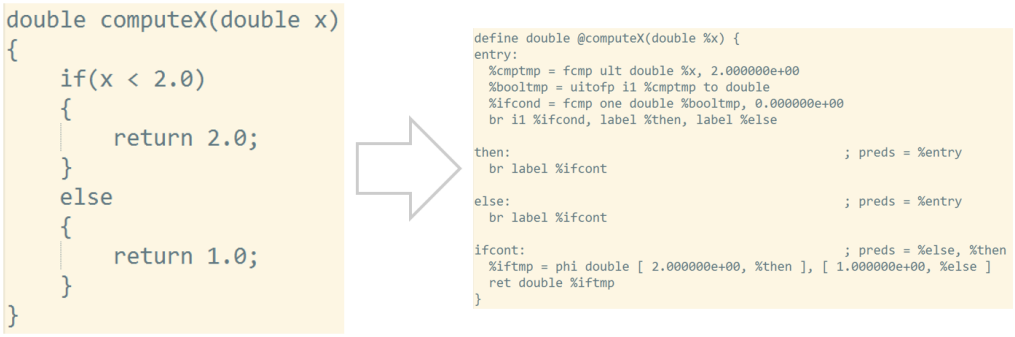


Abbildung 7.6: Umwandlung von IFC-PL In LLVM IR

von virtuellen Maschinen für LLVM-IR-Code[4]. Virtuelle Maschinen können auch fortgeschrittene Techniken wie z. B. Binary Translation unterstützen. Bei dieser Technik werden Codeteile, die häufig ausgeführt werden, direkt in den Befehlssatz der Host-CPU überführt, d. h. der LLVM-IR-Code muss nicht instruktionsweise interpretiert und in Hostinstruktionen übersetzt werden, sondern kann direkt aus einem Code-Cache für vorübersetzte Einheiten abgegriffen und ausgeführt werden. Diese Code-Cache wird üblicherweise dynamisch, abhängig von der Häufigkeit des Aufrufes bestimmter Programmbestandteile, erzeugt. Dabei werden Codeteile, die sehr häufig aufgerufen werden, direkt in den Befehlssatz der Host-CPU übersetzt und im Code-Cache zwischengespeichert. Weitere Details zu dieser Technik können beispielsweise dem Buch (Smith & Nair, 2005) entnommen werden. Dadurch wird die Übersetzung des Maschinencodes der virtuellen Maschine für die Zielplattform nicht zum Flaschenhals und kann ähnlich performant ausgeführt werden wie nativ kompilierter Code. Binary Translation arbeitet auf Ebene des Maschinencodes der virtuellen Maschine bzw. auf Ebene des Befehlssatzes der Ziel-CPU. Bei einer direkten Übersetzung einer Hochsprache in Binärcode für die Ziel-CPU während der Laufzeit eines Programms spricht man von einer Just-in-Time Compilation. Eine Binary Translation ist im Prinzip eine Just-in-Time Compilation auf Ebene des Binärcodes (Maschinencodes).

Auf Basis der Werkzeuge Flex, Bison und LLVM scheint die Implementierung eines Codegenrators für IFC-PL prinzipiell möglich; sie ist jedoch mit erheblichem zeitlichem Entwicklungsaufwand verbunden und wurde deshalb nur für eine kleine Teilmenge der IFC-PL prototypisch umgesetzt.

7.2.3 IFC-PL-Laufzeitumgebung auf Basis eines Transpilers

Ein Source-to-Source-Compiler, Transcompiler oder Transpiler ist eine Art von Übersetzer, der den Quelltext eines in einer Programmiersprache geschriebenen Programms als Eingabe entgegennimmt und daraus den entsprechenden Quelltext in einer anderen Programmiersprache erzeugt. Dabei dient als Zielsprache häufig eine Hochsprache wie beispielswiese die Programmiersprache C++. Die

[4] wie z. B. VMIR (`https://github.com/andoma/vmir`)

Übersetzung der IFC-PL in eine Hochsprache bietet aus Implementierungssicht viele Vorteile, da ein Übersetzer der Zielsprache genutzt werden kann und dadurch viele Implementierungsaufgaben beim Übersetzerbau entfallen, z. B.:

- Eine Speicherverwaltung und Registerallokation auf Assemblerebene entfällt.
- Eine Optimierungsphase auf Binärcodeebene entfällt.
- Es ist kein tiefergehendes Verständnis für die Erzeugung von Maschinencode bzw. einer speziellen CPU-Architektur nötig.
- Durch das höhere Abstraktionsniveau, das eine Hochsprache bietet, lassen sich Sprachelemente wie z. B. Klassen einfacher realisieren

Für die Realisierung der IFC-PL-Laufzeitumgebung, die alle beschriebenen Sprachfeatures aus Kapitel 5 unterstützt, wurde ein Transpileransatz gewählt. Der Quellcode dieses Transpilers steht unter der GPLv3 Lizenz unter (CMS, 2018a) bereit. Der Transpiler selbst ist in der Programmiersprache C++ entwickelt worden und nutzt Flex und Bison zum Lexen und Parsen des IFC-PL-Quelltextes. Ziel dieses IFC-PL-Transpilers ist es, aus einem IFC-PL-Programm ein äquivalentes C++-Programm zu erzeugen. Zudem erzeugt der Transpiler eine IFC-PL-Laufzeitumgebung, die alle Bibliotheksfunktionen der IFC-PL-Standard-Bibliothek (siehe Abschnitt 5.19) bereitstellt. Der generierte C++-Quelltext wird dann mittels eines C++-Compilers, abhängig vom Betriebsmodus (siehe Abschnitt 5.3), in eine ausführbare Datei (Executable) oder eine Bibliothek übersetzt.

Eine so erzeugte IFC-PL-Bibliothek erscheint nach außen hin für einen Benutzer wie eine native C++-Bibliothek. Diese kann dann mittels eines Language-Bindings von verschiedenen Programmiersprachen aus genutzt werden. Beispielsweise stellt das .NET Framework für .NET-Sprachen wie C# oder Visual Basic die DllImportAttribute-Klasse bereit, mit der Methoden attributiert werden können, die ihren Einstiegspunkt in einer Unmanaged Dynamic-Link Library (DLL) besitzen (Troelsen, 2003). Ein weiteres Beispiel für die Anbindung einer C++-Bibliothek an eine Programmiersprache stellt das Java Native Interface (JNI) dar. JNI ist ein Framework, das es ermöglicht, aus einem Java-Programm heraus nativen C++-Code aufzurufen (Liang, 1999). Auch viele andere Sprachen wie z. B. Python stellen Möglichkeiten bereit, nativen C++-Code anzubinden. Eine Übersicht des beschriebenen Übersetzungsprozesses und die Anbindung an eine Hostapplikation (Zielapplikation) ist in der Abbildung 7.7 dargestellt.

Die Überführung eines IFC-PL-Programms in ein C++-Programm beginnt mit der Lexer- und Parserschicht. Als Ergebnis entsteht dabei ein Syntaxbaum. Für jeden Knotentypen dieses Syntaxbaums wurde eine Klasse angelegt. Eine Übersicht dieser Klassen ist in Abbildung 7.8 dargestellt.

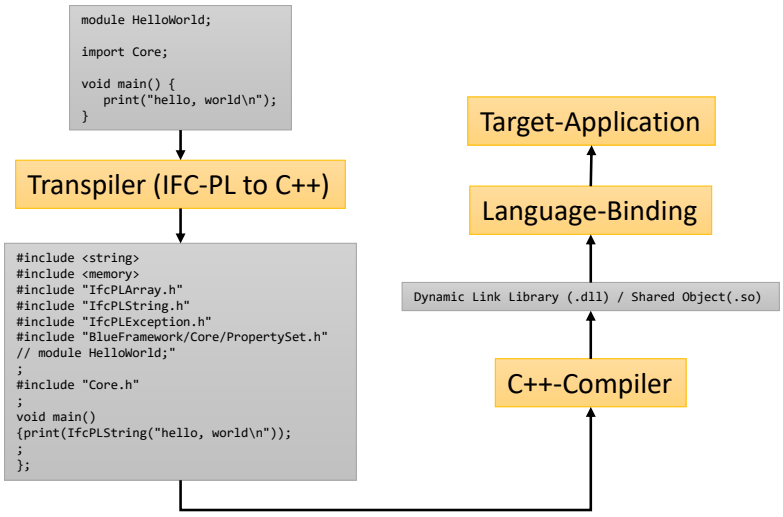


Abbildung 7.7: Anbindung eines Source-to-Source-Translation-Ansatzes an eine Anwendung (Target-Application)

Alle Knoten des Syntaxbaums sind von der Basisklasse Node abgeleitet. Der Knoten für eine bedingte Anweisung wird in der Klasse IfStatement definiert und ist im folgenden Codefragment dargestellt:

```
1  class IfStatement : public Statement {
2  public:
3      std::unique_ptr<Expression> conditionExpr;
4      std::unique_ptr<Statement> thenExpr;
5      std::unique_ptr<Statement> elseExpr;
6
7      IfStatement(std::unique_ptr<Expression> conditionExpr,
8                  std::unique_ptr<Statement> thenExpr,
9                  std::unique_ptr<Statement> elseExpr)
10         : conditionExpr(std::move(conditionExpr)),
11         thenExpr(std::move(thenExpr)),
12         elseExpr(std::move(elseExpr)) {
13     }
14
15     virtual void accept(NodeVisitor *vistor, std::ostream &out) const {
16         vistor->visit(this, out);
17     }
18 };
```

Der bedingte Ausdruck besitzt eine Bedingung (conditionExpr) sowie einen Then- (thenExpr) und Else-Zweig (elseExpr). Der IFC-PL-Transpiler nutzt das Visitor Pattern für den Syntaxbaum (siehe accept-Methode der Klasse IfStatement). Dies ermöglicht es, den Syntaxbaum mit einem Visitor zu besuchen. Der Visitor CppTransform besucht jeden Knoten des Syntaxbaums und generiert für jeden Knoten den entsprechenden äquivalenten C++-Code. Die Codegenerierung für eine If-Anweisung ist im folgenden Quelltextfragment dargestellt:

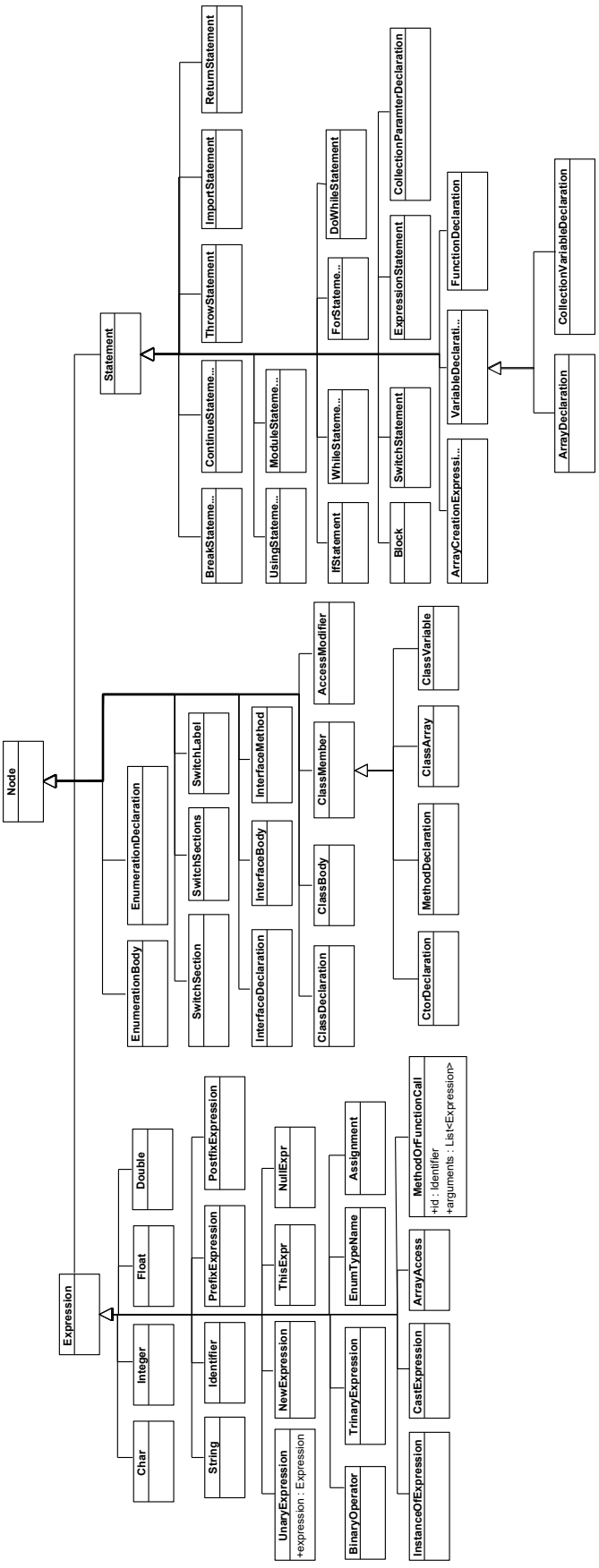


Abbildung 7.8: Knoten im Syntaxbaum

```
1  virtual void CppTransform::visit(const IfStatement* node, std::ostream& out) {
2      out << "if(";
3      node->conditionExpr->accept(this, out);
4      out << ")" << std::endl;
5      out << "{" << std::endl;
6      node->thenExpr->accept(this, out);
7      out << std::endl << "}" << std::endl;
8
9      if (node->elseExpr) {
10         out << "else" << std::endl;
11         out << "{" << std::endl;
12         node->elseExpr->accept(this, out);
13         out << std::endl << "}" << std::endl;
14     }
15 }
```

Die Klasse CppTransform generiert für eine If-Anweisung auf C++-Seite zunächst den C++-Quelltext if. In Klammern gesetzt wird dann die Bedingung der If-Anweisung generiert. Die Bedingung ist in der Variablen conditionExpr vom Typ Expression gespeichert. Diese Klasse besitzt wiederum eine accept-Methode, die es dem Visitor (CppTransform) ermöglicht, den entsprechenden Syntaxknoten zu besuchen. Auf ähnliche Weise wird auch der C++-Code für den Then- und Else-Zweig generiert.

Eine Besonderheit stellt die Import-Anweisung dar. Stößt der Transpiler auf eine Import-Anweisung, die ein EXPRESS-Schema importiert, so muss dieses EXPRESS-Schema ebenfalls geparst und anschließend für jede Typ- und Entity-Deklaration des EXPRESS-Schemas entsprechend ein C++-Typ angelegt werden. Dieser Schritt wurde auf Basis des zuvor vorgestellten Early-Binding-Generators oipExpress realisiert.

Der Transpiler ist auf das Vorhandensein eines C++-Compilers angewiesen. Es gibt frei erhältliche C++-Compiler wie z. B. Dev-C++ (Orwell, 2018). Dev-C++ besitzt eine Größe von ca. 40 Megabytes und könnte relativ leicht zusammen mit dem entwickelten IFC-PL-Transpiler gebündelt und an Anwender ausgeliefert werden.

In der Implementierung des Transpilers fiel die Entscheidung, als Buildsystem für den generierten C++-Code CMake zu verwenden. CMake unterstützt zahlreiche Buildumgebungen wie beispielsweise Visual Studio. Im Rahmen dieser Arbeit wurde Visual Studio 2015 bzw. 2017 verwendet, um den vom Transpiler generierten C++-Code zu übersetzen. In ähnlicher Weise könnte man auch Dev-C++ oder einen beliebigen anderen C++-Compiler in den Buildprozess einbinden.

7.3 Die TUM Open Infra Platform

Die TUM Open Infra Platform (Amann *et al.*, 2016) ist eine Software, deren Fokus auf der Anzeige und Konvertierung von Trassierungsdaten liegt. Diese bietet Unterstützung für eine Teilmenge der folgenden Trassierungsdatenformate:

- LandXML 1.2
- OKSTRA
- IFC Alignment 1.0
- IFC Alignment 1.1/IFC 4.1
- Datenart 40
- Open Street Maps
- IFC-Bridge (Yabuki *et al.*, 2006; Lebegue *et al.*, 2007)
- InfraGML

Daneben bietet die Software noch weitere Möglichkeiten an, u. a. :

- Unterstützung von Querprofilen, Damm- und Einschnittsbauwerken
- Überlagerung mit Kartendiensten (Open Street Maps)
- ifcOWL 4.1/OkstraOWL Export
- Visualisierung einer Untermenge von IFC 2.3 bzw. IFC 4.0 Modellen
- Rendern von Punktewolken im LAS 1.1/1.2 Datenformat
- Umwandlung von Heightmaps in digitale Geländemodelle
- Triangulierung von XYZ-Punktedaten

Abbildung 7.9 zeigt einige Screenshots der TUM Open Infra Platform (kurz OIP), die verschiedene Funktionalitäten der Software darstellen. Die Software wurde hauptsächlich in der Programmiersprache C++ implementiert. Der Quellcode kann von (CMS, 2018c) bezogen und frei unter der GPLv3 Lizenz verwendet werden.

Die OIP gliedert sich in verschiedene Teilpakete, die sich jeweils in unterschiedlichen Sourcecode-Repositories befinden.

– BlueFramework3: Einfaches 3D-Framework, das für die 3D-Visualiserung innerhalb der OIP genutzt wird. Dieses stellt ein Rasterisierungs-Interface zum Rendern von 3D-Objekten bereit. Dieses Interface wird durch verschiedene Render-Systeme umgesetzt, unter anderem durch Direct3D 11 (Zink *et al.*, 2011), Direct3D 12 (Luna, 2016) und OpenGL (Kessenich *et al.*, 2016). Außerdem gibt es eine prototypische Vulkan-Implementierung (Sellers & Kessenich, 2016). Zudem stellt das Framework Möglichkeiten zur Navigation in 3D-Szenen, beispielsweise eine Arcball-Steuerung (Shoemake, 1992) oder einen View-Cube (Khan *et al.*, 2008), bereit. Daneben werden Basisfunktionen für das Laden und Speichern von Bilddaten sowie mathematische Funktionen für den Umgang mit linearer Algebra bereitgestellt. Beispielsweise sind hier Funktionen und Klassen für Vektoren, Matrizen und Quaternionen (Vince, 2011) zu finden. Das BlueFramwork3 ist eine eigenständige, vom Verfasser entwickelte Bibliothek, die auch unabhängig von der OIP lauffähig ist. Jedoch wurde dieses Framework mit starkem Fokus auf die OIP entwickelt. Einige Funktionen der IFC-PL-Laufzeitumgebung, die der IFC-PL-Transpiler ebenfalls bereitstellt, sind auf Basis der mathematischen Funktionen und Klassen des BlueFrameworks realisiert worden.

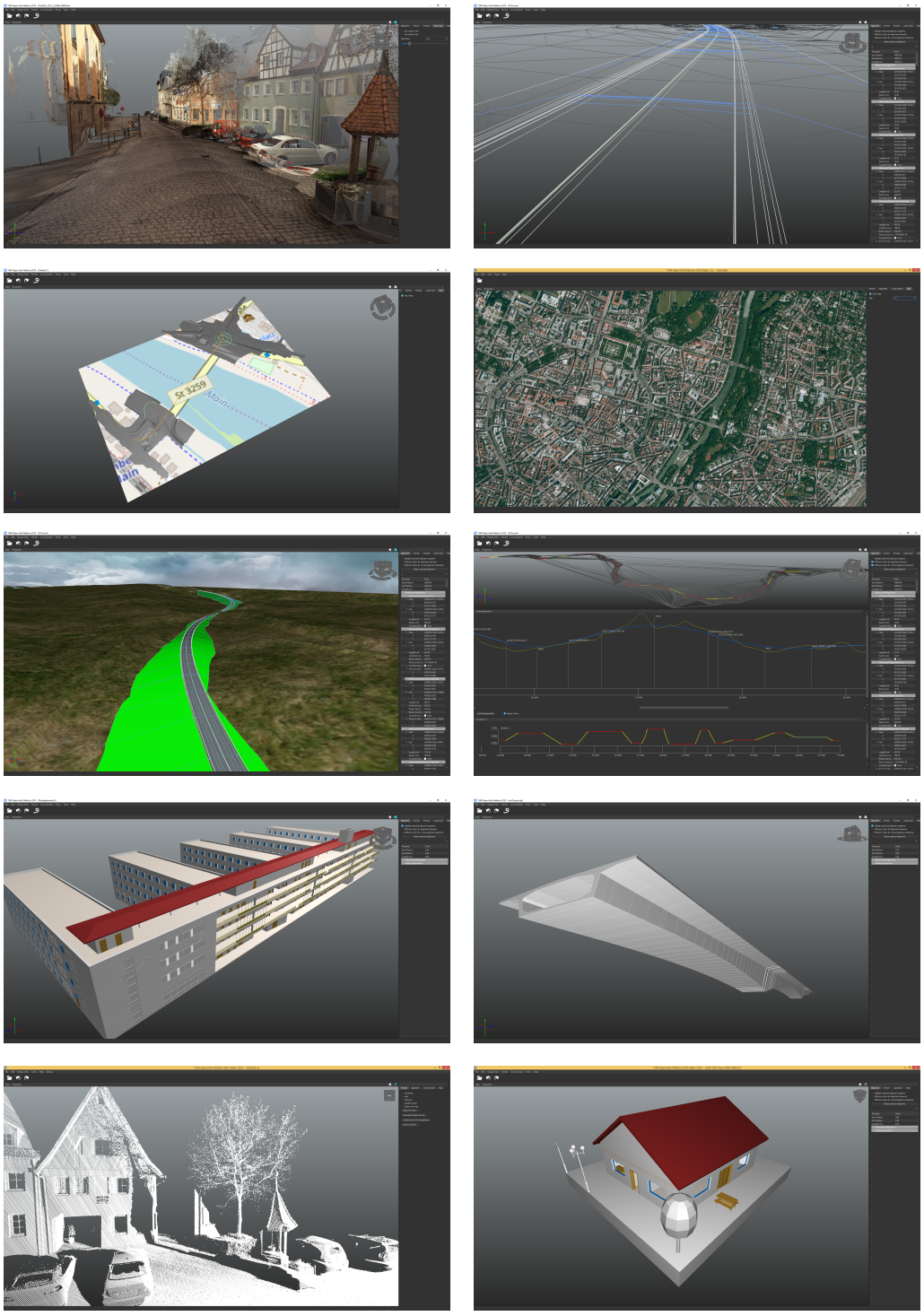


Abbildung 7.9: Verschiedene Screenshots der TUM Open Infra Platform

– OpenInfraPlatform.EarlyBinding: Hierbei handelt es sich um eine Reihe von Early-Bindings, die mittels oipExpress generiert wurden. Im Detail beinhaltet dieses Repository Early-Bindings für IFC 2.3, IFC 4, IFC Alignment 1.0, IFC Alignment 1.1/IFC 4.1, IfcRoad (TUM-Vorschlag zur Erweiterung von IFC Alignment für die Beschreibung von Straßenquerschnitten (Amann *et al.*, 2015)), IfcTunnel (TUM-Vorschlag für Tunnelbauwerke (Jubierre & Borrmann, 2014)) und IfcBridge (französischer und japanischer Vorschlag zur Erweiterung von IFC um Brückenbauwerke (Yabuki *et al.*, 2006)). Die Auslagerung der Early-Bindings in ein eigenes Repository erfolgte, um die Compile-Zeit der grafischen Oberfläche der OIP zu verkürzen. Im Regelfall ändern sich die Bindings nur selten, da Schemaänderungen nicht häufig vorkommen. Beispielsweise wurde das IFC 4.0 Schema ursprünglich 2013 veröffentlicht. 2015 wurde das IFC 4.0 Addendum 1 veröffentlicht, das eine Erweiterung für optimierte Polylinien und Bögen enthielt. Mit dem IFC 4.0 Addendum 2 wurden 2016 erweiterte Boundary Representations eingeführt. Kleinere technische Änderungen wurden 2017 mit dem IFC 4.0 Addendum 2 TC1[5] umgesetzt. Am IFC 4.0 Schema gab es damit innerhalb von ca. vier Jahren nur viermal eine Änderung. Dieses Beispiel zeigt, dass Änderungen am EXPRESS-Schema der IFC relativ selten sind, jedoch ist bei umfangreichen Softwareprojekten wie der OIP öfter ein sogenannter Clean-Build notwendig. Bei einem Clean-Build wird der gesamte Quelltext einer Software neu übersetzt. Spaltet man jedoch ein großes Softwareprojekt in Teilprojekte, so lässt sich dieser Aufwand minimieren. Genau aus diesem Grund wurden die Early-Bindings der OIP in ein eigenes Softwarepaket verlagert. Die Early-Bindings umfassen ca. 9400 *.cpp-Dateien. Ein Clean-Build (Prallel Build) aller Early-Bindings benötigt auf einem Rechner mit einer Intel Core i7 5820K CPU (6 Kerne mit Hyperthreading, d.h. 12 logische Kerne mit 3,3 GHz) mit 16 GB RAM und einer SSD mit 540 MB/s sequentieller Lese- bzw. 520 MB/s Schreibgeschwindigkeit etwa 28 Minuten. Die Early-Bindings sind ein Teil der OIP, welche die längste Compile-Zeit benötigen. Jedoch muss dieses Paket nur bei einer Änderung eines Schemas neu gebaut werden, was relativ selten der Fall ist. Das Early-Bindings Paket hat keine Abhängigkeiten zu anderen Paketen der OIP und kann daher auch eigenständig in anderen Projekten genutzt werden, um beispelsweise IFC 4.1 STEP-Dateien lesen bzw. diese schreiben zu können.

– OpenInfraPlatform.Infrastructure: Dies ist das Herzstück der Open Infra Platform. OpenInfraPlatform.Infrastructure ist eine Bibliothek, die alle relevanten Aufgaben und Funktionen der Open Infra Platform abbildet. Dazu gehören Klassen und Funktionen, die die Verarbeitung von Trassierungsdaten, Straßenquerschnitten, digitalen Geländemodellen und weiteren infrastrukturrelevanten Geometrieelementen ermöglichen. Die Bibliothek unterstützt den Import und Export zahlreicher Datenformate und nutzt dabei u. a. die Bibliothek OpenInfraPlatform.EarlyBinding (für den Import von IFC-basierten Daten), die OKLABI (OKSTRA Klassenbibliothek zum Le-

[5] Dieses Version wurde ebenfalls als ISO 16739-1:2017 standardisiert.

sen und Speichern von OKSTRA-Daten), die LibLAS (für den Import von Punktewolken) und die Raptor RDF Syntax Library (für den Export von Daten, die auf der Web Ontology Language beruhen). Die Bibliothek bietet auch ein Language-Binding für C# und Python (diese werden z. B. in (Singer, 2014) und (Wijnholts, 2016) verwendet). Zudem beinhaltet diese Bibliothek einen rudimentär implementierten IFC-PL-Interpreter.

– OpenInfraPlatform.UI: Hierbei handelt es sich um das grafische Frontend (User Interface) der OIP. Dieses befindet sich im gleichen Repository wie OpenInfraPlatform.Infrastructure, könnte jedoch aus technischer Sicht auch in ein eigenes Repository ausgelagert sein und wird logisch gesehen als eigenes Projekt verwaltet. Die grafische Benutzeroberfläche nutzt die BibliothekenOpenInfraPlatform.Infrastructure und BlueFramework3. Alle Screenshots aus der Abbildung 7.9 zeigen die grafische Benutzeroberfläche der OIP. Es gibt zudem noch eine terminal-/konsolenbasierte Version der OIP. Diese ist in das Unterprojekt OpenInfraPlatform.CommandLineUtilities eingebettet. Mit dieser Anwendung ist es beispielsweise möglich, per Kommandozeile eine OKSTRA-Datei in eine IFC 4.1-Datei umzuwandeln.

Die verschiedenen Softwarekomponenten sind teilweise durch Unit- und Integrationstests abgedeckt. Einige dieser Integrationstests sind grafisch basiert, d. h. diese nehmen einen Screenshot der Anwendung auf und vergleichen diesen mit einem zuvor gemachten Referenzbild, das als korrekt befunden wurde. Sollte es Abweichungen zwischen dem Referenzbild und dem Screenshot geben, so schlägt der Integrationstest fehl. Einige Referenzbilder dieser Integrationstests sind in Abbildung 7.10 dargestellt. Probleme, die beim Testen, basierend auf dieser Technik, auftreten können und entsprechende Lösungen dazu werden z. B. in (Amann *et al.*, 2013) beschrieben.

Die IFC-PL-Umgebung wurde prototypisch im Rahmen dieser Arbeit in die OIP integriert.

7.3.1 IFC Early Binding Meta Template Library

Die verschiedenen IFC-Versionen sind größtenteils gleich und besitzen eine gemeinsame Basis. Beispielsweise gibt es die Entität IfcDoor im IFC-Standard 2.3, 4.0 und 4.1. Die Visualisierung eines Objektes dieser Entität läuft innerhalb der OIP immer gleich ab. Jedoch ist aufgrund des Early-Bindings eine Entität vom Typ IfcDoor aus der IFC-Version 2.3 nicht gleich einer Entität der Version 4.0. Für jede unterschiedliche IFC-Version gibt es eine eigenständig IfcDoor-Klasse, die jeweils in einem entsprechenden Namespace liegt.

Damit man unabhängig von der IFC-Version den gleichen Code für unterschiedliche IFC-Versionen verwenden und gleichzeitig verschiedene IFC-Versionen unterstützen kann, wurde ein Template-basierter Ansatz (ifcEMT) innerhalb der OIP realisiert. Die IFC Early Binding Meta Template Library (ifcEMT) ist Teil der OIP.

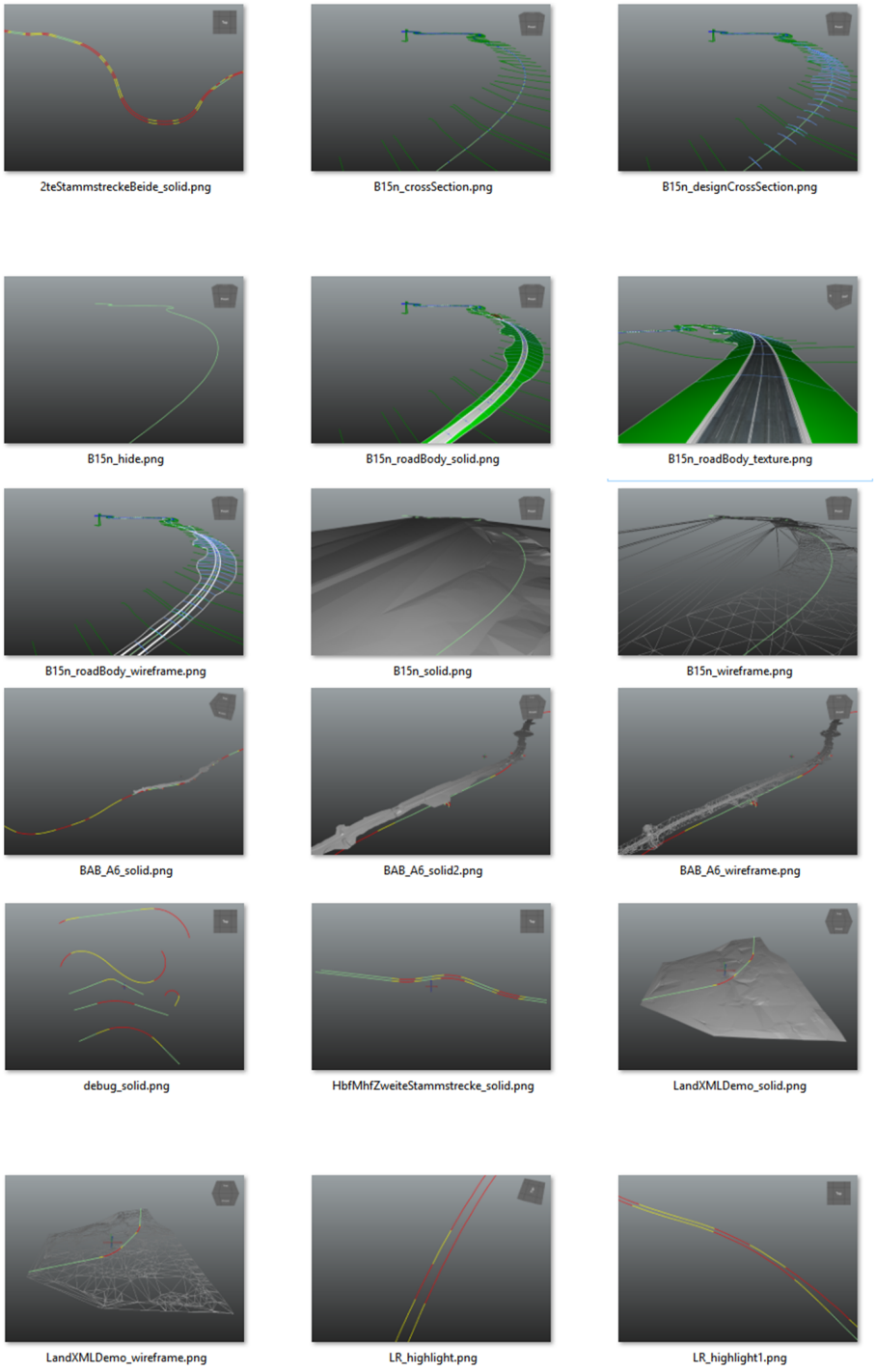


Abbildung 7.10: Verschiedene grafische Integrationstest der OIP

Anhand der folgenden Pseudocodeabschnitte soll die Funktionsweise der ifcEMT verdeutlicht werden. Im folgenden Beispiel wird exemplarisch die Entität IfcFace aus einem IFC 2.3- und IFC 4.0-Early-Binding betrachtet. Die Entität IfcFace taucht in beiden Bindings im entsprechenden Namespace auf:

```
1  namespace OpenInfraPlatform {
2          namespace Ifc2x3
3          {
4                  class IfcFace;
5                   ...
6          }
7  }
8
9  namespace OpenInfraPlatform {
10         namespace Ifc4
11         {
12                 class IfcFace;
13                  ...
14         }
15 }
```

Für jeden Typ einer Entität wird ein Templateparameter in einem Template mit dem Namen IfcEntityTypes angelegt. Dabei beinhaltet dieses Template alle Typen, die sich bei der Vereinigung aller Entitätstypen beider IFC-Versionen ergeben. Da dieses Template mehr als 600 Typen umfasst, wurde es nicht per Hand, sondern automatisch mittels einer speziellen oipExpress Generatorimplementierung erzeugt. Vereinfacht ist das Ausgabeergebnis dieses Prozesses am Beispiel der Entität IfcFace nachfolgend dargestellt:

```
1  namespace emt {
2      template <
3            typename IfcFaceTypeType
4             ...
5      >
6      struct  IfcEntityTypes
7      {
8            typedef IfcFaceTypeType IfcFace;
9             ...
10     };
11 }
```

Die Idee hierbei ist, dass Algorithmen nicht mehr auf Basis von Early-Bindings arbeiten, sondern auf Basis des IfcEntityTypes-Templates. Dadurch ist es möglich, den gleichen Code für verschiedene Early-Bindings zu nutzen. Gleichzeitig können aber auch, wenn Unterschiede im Verhalten zwischen verschiedenen IFC-Versionen vorliegen, diese durch eine C++-Template-Spezialisierung behandelt werden. C++-Template-Spezialisierungen werden verwendet, wenn das standardmäßige Schablonenverhalten (Templateverhalten) nicht greifen kann und stattdessen, abhängig vom konkreten Typ, ein spezifisches Verhalten implementiert werden soll, das vom definierten Templateverhalten abweicht.

Die Verknüpfung eines Early-Bindings mit dem Template erfolgt durch eine typedef-Deklaration:

```
1  namespace emt {
2      typedef IfcEntityTypes<
3          OpenInfraPlatform::Ifc2x3::IfcFace
4          ...
5      >
6      Ifc2x3EntityTypes;
7
8      typedef IfcEntityTypes<
9          OpenInfraPlatform::Ifc4::IfcFace
10         ...
11     >
12     Ifc4EntityTypes;
13 }
```

Algorithmen, die unabhängig von einem konkreten Early-Binding implementiert sind, werden erst, wenn es nötig ist, durch den tatsächlichen Datentyp ersetzt, wie das anschließende Beispiel zeigt:

```
1  ...
2  if (ifcSchema == IfcPeekStepReader::IfcSchema::IFC_2)
3  {
4      OpenInfraPlatform::AsyncJob::getInstance().
5          updateStatus(std::string("Importing Ifc2x3 ").append(filename));
6
7      using namespace OpenInfraPlatform::Ifc2x3;
8      importIfcGeometry <emt::Ifc2x3EntityTypes, UnitConverter, Ifc2x3Model, IfcStepReader,
9          Ifc2x3Exception, Ifc2x3Entity >(tempIfcGeometryModel_, filename);
10 }
11 else if (ifcSchema == IfcPeekStepReader::IfcSchema::IFC_4)
12 {
13     OpenInfraPlatform::AsyncJob::getInstance().
14         updateStatus(std::string("Importing Ifc4 ").append(filename));
15
16     using namespace OpenInfraPlatform::Ifc4;
17     importIfcGeometry<emt::Ifc4EntityTypes, UnitConverter, Ifc4Model, IfcStepReader,
18         Ifc4Exception, Ifc4Entity>(tempIfcGeometryModel_, filename);
19 }
20 ...
```

7.4 Zusammenfassung

In diesem Kapitel wurde ein EXPRESS Early-Binding-Generator (oipExpress) vorgestellt. Dieser kann an spezifische Anforderungen mittels sogenannter Generatoren angepasst werden. Diese können auf ein Metamodell zugreifen, das durch das Einlesen eines EXPRESS-Schemas erzeugt wird.

Im Anschluss daran wurden drei unterschiedliche Realisierungsvarianten der IFC-PL-Laufzeitumgebung skizziert. Hierbei wurde vorgestellt, wie sich die IFC-PL-Laufzeitumgebung in Form eines Interpreters auf Basis einer virtuellen Maschine und auf Basis eines Transpilers realisieren lässt. Der Transpileransatz bot im Rahmen der prototypischen Umsetzung des IFC-PL-Konzepts die größten Vorteile, weshalb auch alle in dieser Arbeit vorgestellten Sprachfeatures der IFC-PL mittels dieser Variante letztlich realisiert wurden. Im Rahmen der Übersetzung mittels Transpiler wird auch der zuvor vorgestellte Early-Binding Generator oipExpress genutzt. Dieser wird im Übersetzungsvorgang verwendet, um Import-Anweisungen, die ein EXPRESS-Schema importieren, zu unterstützen.

Im Rahmen der prototypischen Umsetzung wurde der IFC-PL-Ansatz u. a. in die TUM Open Infra Platform integriert. Im Rahmen der Entwicklung der OIP wurden auch noch weitere Implementierungsarbeiten durchgeführt, beispielsweise die Umwandlung von Trassierungsdaten in unterschiedliche Datenformate. Dadurch ist eine Bewertung der Komplexität von Bauwerksdatenmodellen auf Basis einer Metrik auf Instanzebene möglich (siehe Abschnitt 2.5).

Eine Schwierigkeit bei der Unterstützung unterschiedlicher IFC-Versionen auf Basis eines Early-Bindings stellen die meist gleichnamigen und sehr ähnlich aufgebauten Klassen dar, die zum größten Teil identisch sind. Um hier eine Codeduplizierung zu vermeiden, wurde ein Meta-Template-Ansatz (IFC Early Binding Meta Template Library) vorgestellt, der hilft, eine Duplizierung von Quelltext beim Einsatz von Early-Bindings zu vermeiden.

Kapitel 8

IFC-PL-Anwendungsbeispiele

Dieses Kapitel behandelt drei unterschiedliche Anwendungsfälle, die auf Basis der IFC-PL-Infrastruktur umgesetzt werden: die Beschreibung von beliebigen Übergangskurven (Abschnitt 8.1), die parametrische Geometriebeschreibungen von Volumenkörpern (Abschnitt 8.2) und die Prüfung von Normen und Richtlinien (Abschnitt 8.3). Des Weiteren werden die damit verbundenen Vorteile verdeutlicht.

Das Beispiel der Übergangskurven zeigt, wie durch den IFC-PL-Ansatz flexibel neue Implementierungen von Übergangskurven auf Basis einer zuvor festgelegten Schnittstellenbeschreibung eingebunden und zwischen verschiedenen Anwendungen ausgetauscht werden können. Dieses Beispiel verdeutlich, wie typische Implementierungsdetails, die heute üblicherweise in Standardisierungsgremien von Datenaustauschstandards erarbeitet werden, ausgelagert und durch abstrakte Schnittstellenbeschreibungen ersetzt werden können. Dadurch kann im Rahmen einer Standardisierung auf einem konzeptionell höheren Niveau gearbeitet werden und detaillierte Implementierungsentscheidungen können den Anwendungsentwicklern überlassen werden. Gleichzeitig ist dabei sichergestellt, dass jede Anwendung, welche die Schnittstelle unterstützt, mit beliebigen Realisierungen dieser umgehen kann. Dies ermöglicht es beispielsweise, neue Typen von Übergangskurven einzuführen oder die Parameter, durch die eine Übergangskurve beschrieben wird, zu verändern, ohne einen aufwändigen Standardisierungsprozess für ein Datenaustauschformat durchführen zu müssen. Zudem befreit dieses Vorgehen Anwendungsentwickler davon, jede mögliche Implementierungsvariante einer Übergangskurve umsetzen zu müssen.

Im Abschnitt *Parametrische Geometriebeschreibungen von Volumenkörpern* wird zunächst am Beispiel von parametrischen Profildefinitionen gezeigt, wie der IFC-PL-Ansatz einerseits helfen kann, die Komplexität eines Bauwerksdatenmodells zu reduzieren, und andererseits eine flexible Erweiterung durch neue Profilbeschreibungen ermöglicht. Im Anschluss daran wird die geometrische Beschreibung eines rechtwinkligen Kastenwiderlagers behandelt. Dazu wird zunächst eine Schnittstellenbeschreibung definiert und im Anschluss daran eine Beispielim-

plementierung von Konstruktionsregeln zum Aufbau eines Widerlagers gezeigt. Dabei werden höherwertige Parameter zur Steuerung der Geometrieerzeugung übergeben. Dieses Beispiel zeigt darüber hinaus, wie mittels der IFC-PL intelligente BIM-Objekte definiert werden können, die neben ihren Attributen auch entsprechende Methoden austauschen können, was bisher im Rahmen von konventionellen Ansätzen nicht vorgesehen ist. Dies eröffnet die Möglichkeit, Expertenwissen auf Basis der IFC-PL formal abzulegen und wieder zu verwenden.

Im letzten Teil des Kapitels wird die Prüfung von Normen und Richtlinien betrachtet. Hierbei wird exemplarisch anhand einzelner Regeln, die aus den Richtlinien für die Anlage von Landstraßen (RAL) stammen, gezeigt, wie diese durch die IFC-PL beschrieben und automatisiert überprüft werden können.

8.1 Beschreibung von beliebigen Übergangskurven

In diesem Anwendungsfall wird dargestellt, wie dynamisch zur Laufzeit neue Übergangskurventypen in ein Bauwerksdatenmodell eingeführt und von verschiedenen Fachanwendungen interpretiert werden können, ohne dass diese vorher explizit vereinbart wurden. Als Ausgangsbasis soll hier das Bauwerksdatenmodell IFC 4.1 (Release 4.1.0.4) dienen, das in Abschnitt 2.4.1 beschrieben wird. Dieses wird entsprechend modifiziert, um beliebige Übergangkurven zu unterstützen, die mithilfe der IFC-PL beschrieben werden.

8.1.1 Konventioneller Datenaustausch

Konventionelle Datenmodelle für Trassierungen stellen eine bestimmte Menge von unterstützten Übergangskurven bereit. Eine dynamische Erweiterung der Datenmodelle zur Laufzeit, d. h. nach der Definition eines festen Standards, ist in den konventionellen Ansätzen nicht vorgesehen. Für den Austausch von Trassierungsdaten existieren zahlreiche Datenaustauchformate.

LandXML

Eines der gebräuchlichsten Datenaustauschformate auf internationaler Ebene ist hierbei das LandXML-Datenformat (Rebolj *et al.*, 2008; Ziering *et al.*, 2007). LandXML 1.2 unterstützt 16 verschiedene Typen von Übergangskurven. Dabei stellt das LandXML 1.2 Datenmodell eine Klasse Spiral bereit, die genutzt wird, um Übergangskurven auf generische Weise zu beschreiben. Die Klasse ist schematisch in Abbildung 8.1 dargestellt.

Das Attribut spiType definiert den Übergangsbogentyp, der von einem entsprechenden Spiral-Datensatz beschrieben wird. Der Variablentyp dieses Attributs ist ein Aufzählungsdatentyp, der für jede Übergangsbogenklasse einen entsprechenden Aufzählungswert bereit stellt. Im Prinzip stellt LandXML mit der Klasse Spiral eine generische Schablone für die Erfassung der Daten eines Übergangsbogens bereit. Dabei werden von einer Übergangskurve die entsprechende Länge (length), der Radius am Start- und Endpunkt (radiusStart, radiusEnd), der Über-

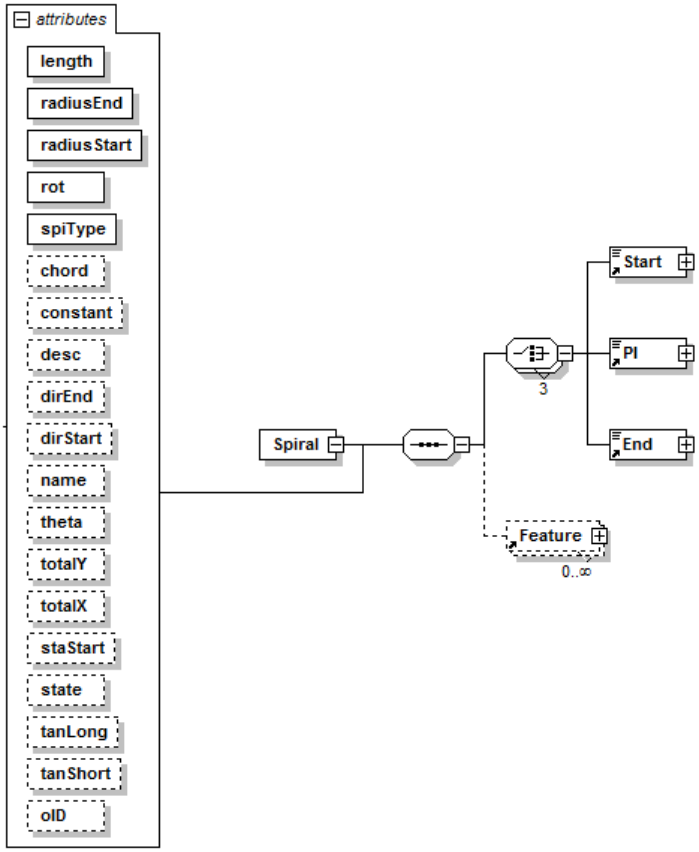


Abbildung 8.1: Schematische Darstellung der Klasse Spiral. Optionale Attribute sind in gestrichelten Rechtecken dargestellt.

gangsbogentyp (spiType), die Drehrichtung (rot), der Start- und Endpunkt sowie der Schnittpunkt der Tangenten am Start- und Endpunkt (PI) als Pflichtattribute erfasst. Daneben können optional auch noch weitere Attribute angegeben werden, z. B. die Start- und Endrichtung (dirStart, dirEnd). LandXML schreibt nicht vor, welche der optionalen Parameter für eine bestimmte Übergangskurve erforderlich sind. Ein exportierendes Programm hat hier freie Hand und kann beliebig viele der vordefinierten optionalen Parameter exportieren. Das Datenformat stellt dabei aber nicht die Datenintegrität, d. h. die Richtigkeit und Konsistenz der Daten, sicher. Beispielswiese könnte bei einer Klothoide als Übergangskurve die entsprechende Klothoidenkonstante (constant) aus den Parametern Länge (length), Start- und Endradius (radiusStart, radiusEnd) berechnet werden. Sollte der berechnete Wert von der gespeicherten Klothoidenkonstante abweichen, so sind die Daten inkonsistent. Zudem wäre in diesem Fall die zusätzliche Angabe der Klothoidenkonstante redundant, da man diesen Wert aus den vorhandenen Werten berechnen kann. Da weiterhin nicht klar ist, welche optionalen Parameter eine bestimmte exportierende Software schreibt, müssen zudem noch alle möglichen Importvarianten von einer Software, die die entsprechenden Daten liest, implementiert und verstanden werden. Dies macht den generischen Ansatz, der von LandXML umgesetzt wird, zwar relativ flexibel, was jedoch durch erhöhten Implementierungsaufwand und die Gefahr von redundanten und inkonsistenten Daten erkauft wird.

LandXML bietet keine Möglichkeit, einen neuen Übergangskurventyp zu beschreiben. Um einen neuen Überganskurventyp in LandXML zu unterstützen, müssten der Aufzählungsdatentyp der verfügbaren Übergangskurventypen entsprechend erweitert und gegebenenfalls weitere Pflichtattribute bzw. optionale Attribute eingeführt werden. Weiterhin müsste diese Änderung in den Standard eingepflegt und entsprechend von den Softwareherstellern umgesetzt werden. Beachtet man dabei, dass zwischen den Veröffentlichungen der LandXML-Version 1.0 und 1.1 ca. vier Jahre liegen und zwischen der Version 1.1 und 1.2 weitere zwei Jahre vergangen sind, wird klar, dass eine Änderung bzw. Erweiterung des Datenmodells nicht sofort zur Verfügung steht, sofern diese überhaupt akzeptiert und umgesetzt wird.

OKSTRA

OKSTRA 2.017, der beginnend vom Straßenentwurf bis hin zur Bestandsdokumentation und zur Erfassung von Verkehrsdaten genutzt wird, unterstützt nur Klothoiden als Überganskurven. Abbildung 2.11 zeigt die Umsetzung von Klothoiden als Übergangskurven im OKSTRA-Standard. Eine Klothoide wird als Achselement umgesetzt. Dieses besitzt einen Achselementtyp, der durch ein Attribut Langtext den genauen Achselementtyp spezifiziert. Auch OKSTRA bietet in der aktuellen Version keine Möglichkeit, neue Übergangskurventypen zu definieren, die ohne Änderung des Datenformats bzw. des Standards genutzt werden können. Theoretisch besteht hier die Möglichkeit, mittels eines Änderungsantrags einen Vorschlag für einen neuen Übergangskurventyp einzureichen. Dieser wird von der OKSTRA-Pflegestelle bearbeitet und Experten vorgelegt, die entscheiden, ob die

gewünschte Änderung in das Datenmodell einfließen kann. Auch dieser Prozess ist zeitaufwändig und muss im Anschluss auch erst noch softwareseitig unterstützt werden (vgl. Abschnitt 4.2). Im Gegensatz zu LandXML gibt es zwar einen formalen Prozess für die Beseitigung von Fehlern wie zur Erweiterung oder Änderung des fachlichen bzw. informationstechnischen Umfangs des OKSTRA-Standards, jedoch kann eine Umsetzung eines Vorschlags auch eine längere Zeit in Anspruch nehmen. Beispielsweise sind zwischen der Veröffentlichung der Version 2.016 und 2.017 mehr als zwei Jahre vergangen. Prinzipiell besteht auch hier das Problem, dass sich das Datenmodell nicht dynamisch zur Laufzeit erweitern lässt.

IFC

IFC 4.1 kennt sechs verschiedene Übergangskurventypen. Darunter der S-förmige Übergangsbogen (Übergangsbogen nach Schramm bzw. Helmert), die Bloss-Kurve, die Klothoide, Sinuskurve/sinusförmiger Übergangsbogen (nach Klein), Cosinusoide und die kubische Parabel. Dabei wird ebenfalls versucht, die Parameter verschiedener Übergangskurven mithilfe eines generischen Templates zu beschreiben. Abbildung 8.2 zeigt einen Ausschnitt aus der Umsetzung von Übergangskurven des IFC 4.1-Standards.

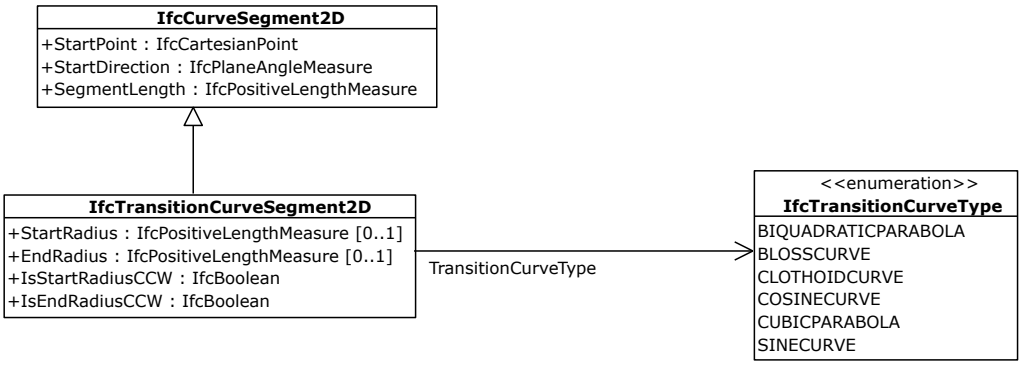


Abbildung 8.2: UML-Klassendiagramm, das die Umsetzung von Übergangskurven in IFC 4.1 zeigt. Die Attribute StartRaidus und EndRaidus sind optional.

Bei einer Klothodie in IFC 4.1 müssen IsStartRadiusCCW und IsEndRadiusCCW immer den gleichen Wert besitzen. Die Klasse IfcTransitionCurveSegment2D sichert nicht per se die Datenkonsistenz, d. h. die Richtigkeit der Daten. Theoretisch könnte ein exportierendes Programm das Attribut IsStartRadiusCCW auf false setzen und das Attribut IsEndRadiusCCW auf true, was zu einem nicht konsistenten Datensatz führen würde. Natürlich könnte man diese Randbedingungen durch eine Regel im EXPRESS-Schema abfangen, jedoch wird dies im veröffentlichten EXPRESS-Schema von IFC 4.1 nicht so umgesetzt. Außerdem unterstützen nicht alle importierenden Anwendungen eine Validierung von EXPRESS-Regeln. Abgesehen von diesen technischen Limitierungen, die sich relativ einfach beheben lassen, wurde ursprünglich eine konsistente Klothoidendarstellung entwickelt. IFC 4.1 wurde in zwei Teilprojekten mit den Namen IFC Alignment 1.0 und IFC Alignment 1.1 um Übergangskurven erweitert. Im ersten Teilprojekt wurden nur Klothoiden als Übergangskurven unterstützt. Für diese wurde eine in sich konsi-

stente Beschreibung gewählt, d. h. es gab keine redundanten Daten, die sich aus den vorhandenen Daten berechnen ließen, und somit waren Klothoiden, die nach dem Vorschlag in IFC Alignment 1.0 entwickelt wurden, immer konsistent per se. Dieser Ansatz würde auch die Einführung von zusätzlichen EXPRESS-Regeln für die Sicherung der Datenintegrität unnötig machen, jedoch für jede Übergangskurve eine eigene Klasse im Datenmodell fordern. Um Änderungen im IFC-Standard innerhalb von buildingSMART durchsetzen zu können, gilt als Grundsatz, die Anzahl der Klassen nicht unnötig weiter zur erhöhen, da hier oft eine Erhöhung der Klassenanzahl gleichgesetzt wird mit einer Erhöhung der Komplexität des Datenmodells. Dies trifft zwar teilweise im Fall der Übergangskurven zu, jedoch geht dies zulasten der Datenkonsistenz. Abbildung 8.3 zeigt den Vorschlag aus dem IFC Alignment 1.0 Projekt für Klothoiden.

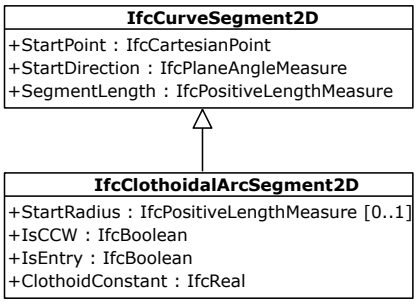


Abbildung 8.3: Um die Datenintegrität zu bewahren, sollte das Datenmodell bereits sicherstellen, dass nur konsistente Daten abgelegt werden können. Die Klasse IfcClothoidalArcSegment2D kommt diesem Designziel schon sehr nahe. Jedoch besitzt diese einen kleinen Schönheitsfehler: Die Klothoidenkonstante sollte nur positive Werte annehmen können. Davon wird zwar implizit ausgegangen, jedoch könnte man dies durch einen geeigneten Datentyp auch erzwingen und hätte dann eine tatsächlich konsistente Darstellung.

Eine Erweiterung des IFC 4.1 Standards um weitere Übergangskurvendatentypen würde erfordern, das Schema entsprechend abzuändern. Dies muss erst von buildingSMART akzeptiert und entsprechend von den Softwareherstellern umgesetzt werden. Derzeit ist IFC 2x3, das 2007 veröffentlich wurde, noch weit verbreitet. Erst allmählich wird Unterstützung auch für IFC 4.0 geboten, das im Jahre 2007 veröffentlich wurde. Von IFC 4.0 zu der Infrastrukturerweiterung in IFC 4.1 sind vier Jahre vergangen. Dies macht deutlich, dass Änderungen am Schema und die damit verbundene Adaptierung durch Softwareentwickler zeitaufwändig sind.

Andere Formate und Forschungsarbeiten

Neben LandXML fehlt auch in anderen Standards wie RoadXML (Chaplier *et al.*, 2012; Ducloux & Millet, 2009), JHDM (Japan Highways Data Model) oder TransXML (Scarponcini, 2006) die Möglichkeit, Daten in Form von algorithmischen Beschreibungen zu transportieren. Viele weitere Trassierungsmodelle, die im Rahmen von Forschungsprojekten entwickelt wurden, wie IFC-Bridge (Yabuki *et al.*, 2006; Lebegue *et al.*, 2007; Ji *et al.*, 2013) oder IFC-Tunnel (Hegemann *et al.*, 2012; Yabuki, 2009; Amann *et al.*, 2013; Vilgertshofer *et al.*, 2016), enthalten ebenfalls keine algorithmischen Beschreibungen für Übergangskurven.

Zusammenfassend lässt sich festhalten, dass etablierte Ansätze keine dynamischen Erweiterungen des Datenmodells bezüglich Übergangskurven zur Laufzeit vorsehen.

8.1.2 Nachteile konventioneller Ansätze

Die konventionelle Vorgehensweise, bei der eine bestimmte (umfangreiche) Menge von verschiedenen Übergangskurventypen manuell in das Datenmodell aufgenommen wird, hat eine Reihe von Nachteilen:

- In einer Region bzw. einem Land oder für ein spezifisches Anwendungsszenario (z. B. Einschienenbahn) könnte eine Überganskurve benötigt werden, die nicht im Datenschema berücksichtigt wurde und daher nicht definiert ist. Die Dresdener Verkehrsbetriebe verwenden beispielweise als Übergangkurve eine Radioide, eine spezielle Übergangsbogenform, die kaum in anderen Bereichen Anwendungen findet (Leitzke, 2017). Als anderes Beispiel kann der Wiener Bogen angeführt werden, der hauptsächlich im österreichischem Bahnbau Verwendung findet. Solche Trassierungselemente können beispielsweise nicht mit dem IFC 4.1-Modell repräsentiert werden. Die dazu nötige Schemaerweiterung und der damit einhergehende Standardisierungsprozess sind sehr langwierig.

- Ein spezifischer Algorithmus für die Interpretation jedes Übergangkurventyps muss implementiert werden. Dies ist zeit- und kostenintensiv für die Softwarehersteller, die deshalb oft die Adaptierung eines erweiterten Standards wie z. B. IFC 4.1 scheuen. Aufgrund des hohen Aufwands implementieren einige Softwarehersteller evtl. nur eine Teilmenge der definierten Kurven bzw. Kurvenparameter, was zu Inkompatibilitäten zwischen verschiedenen Softwareanwendungen führen kann.

- Die Parameter einer bestimmten Übergangskurve können von einem Softwarehersteller fehlinterpretiert und entsprechend falsch in einer bestimmten Fachanwendung umgesetzt werden.

- Verschiedene Stakeholder müssen sich auf eine bestimmte Repräsentation einer Übergangskurve einigen. Beispielsweise gab es bei der Entwicklung von IFC Alignment einen Dissens zwischen dem japanischen buildingSMART-Chapter und den restlichen buildingSMART-Chapters bei der Behandlung von Tangentialpunkten. Das japanische Chapter wollte diese explizit in der Repräsentation speichern, wurde aber letztendlich von den restlichen Chapters überstimmt. Jedoch zeigt dies, dass, abhängig vom konkreten Anwendungsfall, manchmal auch unterschiedliche Repräsentationen der gleichen Information gewünscht sind, diese aber mit einem statischen Modell nicht immer erfüllt werden können. Das japanische buildingSMART-Chapter wies im Rahmen der Erarbeitung von IFC Alignment darauf hin, dass die aktuellen japanischen Bestandsdaten alle einen PVI (Point of Vertical Intersection) besitzen, der von IFC Alignment nur implizit erfasst wird. Hierbei

gab es Befürchtungen, dass aufgrund von numerischen Instabilitäten der berechnete PVI minimal vom explizit in den Bestandsdaten abgespeicherten PVI abweichen könnte. Rechenungenauigkeiten, die bei der Konvertierung in verschiedene Darstellungsarten von Trassierungsdaten entstehen können, werden exemplarisch in (Amann & Borrmann, 2015b) beschrieben. Besitzen die Bestanddaten ein Attribut A, so will man dieses Attribut normalerweise unverfälscht erhalten. Rechnet man dieses in die Repräsentation des statischen Datenmodells um, so kann es evtl. zu Datenverlusten (Fließzahlenungenauigkeit, usw.) kommen.

8.1.3 Umsetzung auf Basis der IFC-PL

Um die genannten Probleme zu lösen, bietet sich die Umsetzung mittels der IFC-PL-Infrastruktur an.

8.1.3.1 Definition einer Schnittstelle

Anstatt wie bisher in einem konventionellen Ansatz festzulegen, welche Daten für die Erfassung einer Übergangskurve nötig sind, einigt man sich auf eine Schnittstelle und nicht auf eine konkrete Implementierung („Program to an interface, not an implementation“). Dies befreit ein Standardisierungskomitee zur Definitionszeit eines Standards von der Einigung auf spezifische Implementierungsdetails, insbesondere auf die Festlegung von bestimmten Parametern, die für eine Übergangskurve gespeichert werden sollen. Dabei wird der Fokus nicht auf die konkrete Umsetzung gelegt, z. B. ob der Startradius oder etwa die Startkrümmung für eine Klothoide gespeichert werden soll, sondern darauf, welche Anforderungen an das Datenmodell gestellt werden. Dabei kann man beispielsweise festlegen, dass es möglich sein muss, den Radius und die Krümmung am Startpunkt der Klothoide berechnen zu können. Diese Anforderungen können dann mithilfe einer Schnittstellenbeschreibung formalisiert werden. Konkret könnte daraus ein Interface entstehen, das Methoden zum Ermitteln des Radius bzw. der Krümmung am Startpunkt bereitstellt. Durch diese Vorgehensweise entkoppelt man sich von der Frage, wie das Datenmodell konkret implementiert werden soll, und fokussiert sich auf die Frage, was das Datenmodell leisten soll. Dabei spielt es dann am Ende keine Rolle, ob eine konkrete Umsetzung (Implementierung) der Schnittstelle mit dem Startradius oder Krümmungswert am Startpunkt arbeitet und jeweils den anderen Parameter daraus ableitet. Zudem gibt es im Rahmen einer Umsetzung des Interfaces die Möglichkeit, den Startradius als Hauptattribut zu wählen und den Krümmungswert am Startpunkt als abgeleitete Größe zu betrachten oder es genau umgekehrt zu machen. Im Rahmen des IFC Alignment Projekt 1.0 und 1.1 war den Projektbeteiligten offensichtlich klar, dass Übergangskurven unterstützt werden müssen und welche Anforderungen an diese gestellt werden. Jedoch gab es intensive Diskussionen, sobald darüber entschieden werden sollte, auf welche Weise diese konkret zu implementieren sind. Bei der Festlegung einer Schnittstelle entfallen diese langwierigen Detailentscheidungen und werden den Softwareentwicklern überlassen. Abbildung 8.4 zeigt einen Vorschlag für eine Schnittstelle einer Übergangskurve.

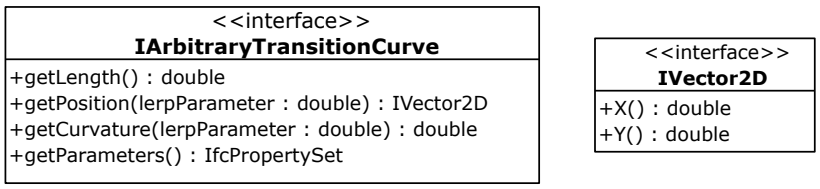


Abbildung 8.4: UML-Klassendiagramm, das die Schnittstelle für eine Übergangskurve zeigt. Es könnten genauso gut noch Methoden zur Ermittlung des Startradius usw. eingeführt werden. Hier soll nur ein minimalistisches Beispiel gezeigt werden, das für den einfachen Anwendungsfall einer Visualisierung ausreichend ist.

Die Schnittstelle wird formal mithilfe der IFC-PL (*IArbitraryTransitionCurve.ifcpl*) wie folgt beschrieben:

```
1  import IFC4X1;
2
3  inteface IVector2D {
4      double X() const;
5      double Y() const;
6  }
7
8  interface IArbitraryTransitionCurve {
9      double getLength() const;
10     IVector2D getPosition(const double lerpParameter) const;
11     double getCurvature(const double lerpParameter) const;
12     IfcPropertySet getParameters() const;
13 }
```

Listing 8.1: Schnittstelle für eine Übergangskurve

Der Datentyp IfcPropertySet wird im IFC 4.1-EXPRESS-Schema definiert, das durch das Import-Statement in der ersten Zeile in der Schnittstellendefinition (siehe Listing 8.1) importiert wird.

Die Schnittstelle bietet folgende Methoden an:

- Berechnung der Länge eines Übergangskurvensegments
- Berechnung der x-, y-Koordinaten für einen gegebenen Interpolationsparameter
- Berechnung der Krümmung für einen gegebenen Interpolationsparameter
- Ermittlung aller Kurvenparameter als IfcPropertySet

Im Regelfall reichen diese Informationen aus, um eine Übergangskurve zu visualisieren und die kartesischen Koordinaten zu einem gegebenen Stationierungswert zu ermitteln, was bei Elementen, die linear zur Trassierung referenziert werden, wie z. B. Straßenquerschnitten, genutzt werden kann. Eine Besonderheit in der Schnittstellendefinition stellt der lerpParameter dar. Diese Variable hat einen Wertebereich von $[0; 1]$. Die Abkürzung lerp steht für „**l**inear int**erp**olation“, und entsprechend wird auf dem Übergangsbogen linear zu seiner Länge eine Position ermittelt. Ein lerpParameter-Wert von 0 liefert die Position des Startpunkts. Entsprechend liefert ein lerpParameter-Wert von 1 die Position des Endpunkts. Der lerpParameter-Wert $0,5$ liefert genau die Position, die man erreicht, wenn man

die halbe Länge, ausgehend vom Startpunkt, auf dem Übergangsbogen zurückgelegt hat. Andere lerpParameter-Werte liefern immer die Position, die erreicht wird, wenn man im Verhältnis zur Gesamtlänge des Bogens den entsprechenden lerpParameter-Wert zurückgelegt hat.

Die gezeigte Schnittstelle (Listing 8.1) ist auf Schemaebene angesiedelt und wird zusammen mit dem EXPRESS-Schema allen Applikationen, die diese Schnittstellenerweiterung verwenden wollen, bekannt gemacht.

8.1.3.2 EXPRESS-Schemaerweiterung

Die Beschreibung einer beliebigen Übergangskurve muss im EXPRESS-Datenschema bekannt gemacht werden. Das IFC 4.1 EXPRESS-Schema wird wie folgt erweitert:

```
1 ENTITY IfcPLInterfaceRealizationConstructionDescription;
2     ClassName : STRING;
3     InitializationData  : IfcPropertySet;
4 END_ENTITY;
5
6 ENTITY IfcArbitraryTransitionSegment2D
7 SUBTYPE OF (IfcCurveSegment2D);
8     ConstructionDescription : IfcPLInterfaceRealizationConstructionDescription;
9 END_ENTITY;
```

Diese Schemaerweiterung ist nur einmalig durchzuführen und erlaubt dann zur Laufzeit die dynamische Erweiterung um beliebige Übergangskurven. Instanzen der Klasse IfcArbitraryTransitionSegment2D können nun verwendet werden, um beliebige Übergangskurven mittels der IFC-PL zu beschreiben. Die Klasse IfcArbitraryTransitionSegment2D definiert selbst nur ein Attribut (ConstructionDescription) vom Typ IfcPLInterfaceRealizationConstructionDescription. Dieses Attribut speichert den Namen einer IFC-PL-Klasse (ClassName), welche die Schnittstelle IArbitraryTransitionCurve realisiert. Beispielsweise kann hier der Klassenname BlossCurve als Wert gespeichert werden. Dies bedeutet für das empfangende Programm, dass die Übergangskurve mit der IFC-PL-Klasse BlossCurve beschrieben wird. Der Quellcode zu dieser Klasse muss sich in einer gleichnamigen Datei mit der Endung .ifcpl befinden. Im Beispiel der Klasse BlossCurve muss der Dateiname BlossCurve.ifcpl lauten. Die Datei BlossCurve.ifcpl, die die entsprechende Implementierung des Interfaces für die Bloss-Kurve enthält, wird zusammen mit der normalen STEP-P21-Datei beim Datenaustausch übergeben. Das Attribut InitializationData der Klasse IfcPLInterfaceRealizationConstructionDescription wird zur Initialisierung des Konstruktors der Implementierung einer Übergangskurve, etwa der Klasse BlossCurve, genutzt. Jede Schnittstellenimplementierung muss einen Konstruktor bereitstellen, der eine Instanz vom Typ IfcPropertySet entgegennimmt. Dieses Property-Set kann beliebige Werte wie z. B. einen Startradius enthalten. Die genaue Anzahl und die Typen der Parameter sind aber dem Implementierer der Schnittstelle überlassen. Dadurch wird die Trennung zwischen Schnittstelle und Implementierung erreicht.

Abbildung 8.5 zeigt eine Übersicht der Schemaerweiterung. Als Ausgangsbasis für das erweiterte Schema wurde das IFC 4.1-EXPRESS-Schema verwendet. Die Erweiterung bietet Unterstützung für beliebige Übergangskurven, die mithilfe der IFC-PL beschrieben werden.

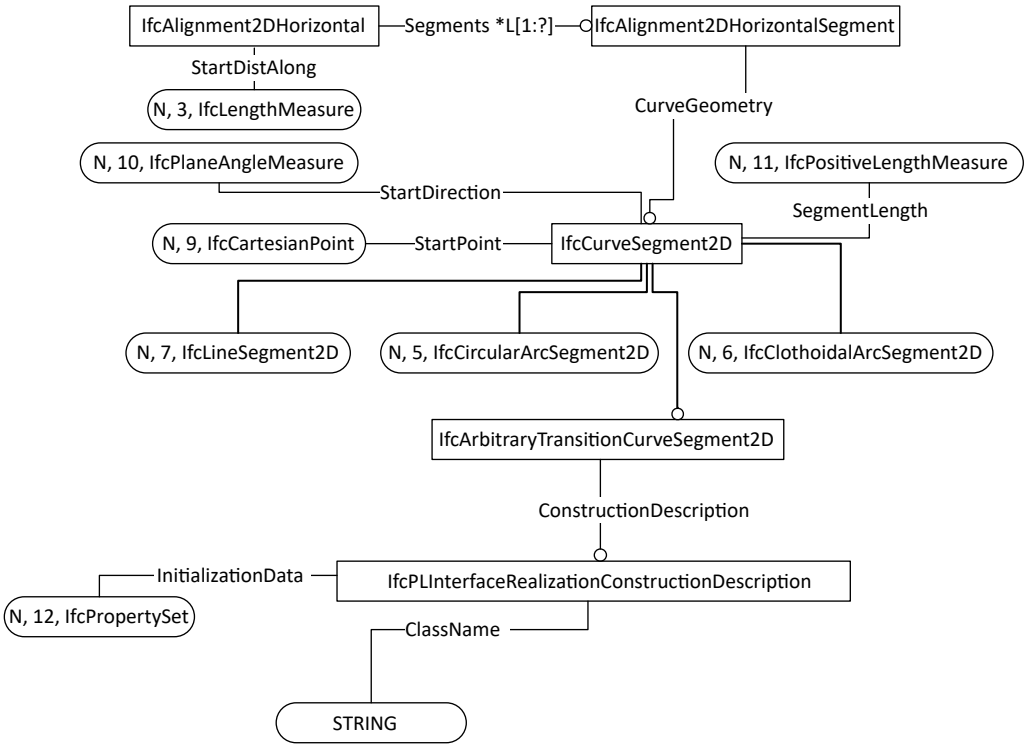


Abbildung 8.5: Ein EXPRESS-G-Diagramm, das die Erweiterung des IFC 4.1-Schemas um beliebige Übergangskurven zeigt. Die Entität IfcArbitraryTransitionSegment2D steht in einer Vererbungsbeziehung zur Klasse IfcCurveSegment2D, die einzelne Segemente des Lageplans beschreibt.

Anzumerken ist hierbei, dass die Entität IfcPLInterfaceRealizationConstruction-Description nicht nur von der Klasse IfcArbitraryTransitionSegment2D genutzt, sondern auch in anderen Anwendungsfällen verwendet werden kann (siehe hierzu z. B. Abschnitt 8.2). Das Schema wurde durch die Veränderung nicht deutlich in seinem Umfang vergrößert, jedoch flexibel für die Erweiterung um weitere Übergangskurven zur Laufzeit erweitert. Allerdings muss dazu zusätzlich eine Schnittstelle (IArbitraryTransitionCurve[1]) vereinbart (siehe Listing 8.1) werden. Diese wird nicht explizit vom EXPRESS-Schema definiert, sondern muss zusätzlich z. B. in einer ifcpl-Datei (IArbitraryTransitionCurve.ifcpl) vorgehalten werden. Diese Schnittstellenerweiterung muss auf Schemaebene vereinbart werden, also zuvor zwischen den am Datenaustausch Beteiligten verhandelt und eindeutig festgelegt werden. Dadurch lassen sich dann beim Datenaustausch dynamisch neue Übergangskurven, die die Schnittstellenbeschreibung erfüllen, hinzufügen. Weiterhin könnte man

[1]Der Name des Interfaces ist beliebig. Dieser könnte prinzipiell ebenfalls IfcArbitraryTransitionCurveSegment2D lauten, wurde aber anders gewählt, damit dieser nicht mit dem Namen der Entität IfcArbitraryTransitioonCurveSegment2D verwechselt wird.

die Klasse IfcClothoidalArcSegment2D, die Klothoiden repräsentiert, und die Klasse IfcTransitionCurveSegment2D, die, basierend auf einem generischen Template, Übergangskurven beschreibt, aus dem Schema entfernen. In der Folge würde man letzten Endes mehr Entitäten, Typen und Attribute aus dem Schema entfernen als hinzufügen.

8.1.3.3 Datenaustausch: Export

Eine Anwendung, welche die eingeführte Erweiterung für Übergangskurven nutzen will, muss eine Instanz der IfcArbitraryTransitionSegment2D-Entität exportieren. Diese erbt die Attribute StartPoint, SegmentLength und StartDirection von der Basisklasse IfcCurveSegment2D. Für dieses Beispiel nehmen wir an, dass unsere selbstdefinierte Übergangskurvenbeschreibung noch auf zusätzliche Parameter angewiesen ist, im Detail auf die Parameter startCurvature, counterClockwise, clothoidConstant und entry. Diese Parameter werden frei vom Implementierer der Übergangskurve (Anwendungsentwickler der Exportfunktionalität) festgelegt und können später von einem IFC-PL-Programm gelesen und verarbeitet werden. Da das Datenmodell diese Daten nicht vorsieht, spezifizieren wir diese Daten mithilfe eines Property-Sets. Dieses enthält anschließend alle selbstdefinierten Parameter samt ihrer Werte und ihres Datentyps. Ein beispielhafter Ausschnitt einer Ausgabe eines exportierenden Programms im STEP-P21-Format ist im folgenden Listing zu sehen:

```
1 #100=IFCPROPERTYSET($,$,'Initialization Parameters',$,(#101,#102,#103,#104));
2 #101=IFCPROPERTYSINGLEVALUE('startCurvature',$,0.0,$);
3 #102=IFCPROPERTYSINGLEVALUE('counterClockwise',$,.T.,$);
4 #103=IFCPROPERTYSINGLEVALUE('clothoidConstant',$,19.570000000000000284,$);
5 #104=IFCPROPERTYSINGLEVALUE('entry',$,.T.,$);
6 #105=IFCPLINTERFACEREALIZATIONCONSTRUCTIONDESCRIPTION(Clothoid,#100);
7 #106=IFCCARTESIANPOINT((−243.77199999999999136,−3.0327000000000001734));
8 #107=IFCARBITRARYTRANSITIONSEGMENT2D(#106,5.07,12.43,#105);
```

Listing 8.2: Exportiertes Property-Set im STEP-P21-Format

Die Entität IfcPLInterfaceRealizatonConstructonDescription speichert nicht nur das Property-Set, das die zusätzlichen Parameter enthält, sondern merkt sich auch den Namen der Klasse, die die Umsetzung des Interfaces realisiert. In diesem Fall lautet der Name der Klasse Clothoid. Abbildung 8.6 zeigt das Beispiel aus dem Listing 8.2 zur besseren Übersichtlichkeit als UML-Objektdiagramm.

Zusätzlich muss ein exportierendes Programm nun auch die Klasse Clothoid als IFC-PL-Programm zusammen mit dem STEP-P21-Daten liefern. Dazu wird eine Datei mit dem Namen der Klasse und der Endung ifcpl geschrieben und zusammen mit der STEP-P21-Datei ausgegeben. Theoretisch könnte man das IFC-PL-Programm selbst auch wieder einfach als IfcText in die entsprechende STEP-P21-Datei schreiben, jedoch wurde zugunsten einer besseren Übersichtlichkeit eine Trennung zwischen dem IFC-PL-Quelltextprogrammen und den normalen STEP-P21-Daten eingeführt. In dem Beispiel der Klothoide wird das entspre-

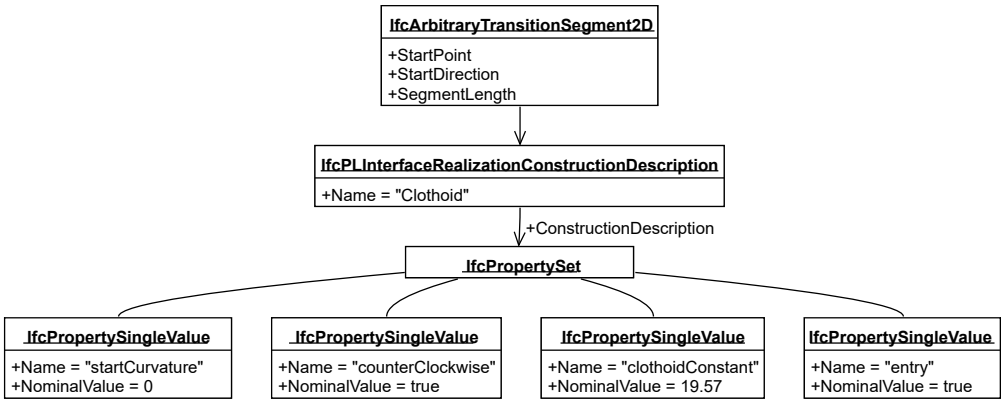


Abbildung 8.6: UML-Objektdiagramm: Die Entität IfcPLInterfaceRealizatonConstructonDescription speichert nicht nur das Property-Set, das die zustäzlichen Parameter enthält, sondern merkt sich auch den Namen der Klasse, die die Umsetzung des Interfaces realisiert

chende Klothoiden-Programm in eine Datei namens Clothoid.ifcpl geschrieben, die wie folgt strukturiert ist[2]:

⟨*Clothoid.ifcpl*⟩≡
```
1    ⟨Clothoid module and import statements 141⟩
2    ⟨IVector2D implementation 143⟩
3    ⟨IArbitraryTransitionCurve implementation 143⟩
```

Die Klasse Clothoid befindet sich im gleichnamigen Modul Clothoid, entsprechend wird dies mit der Modul-Anweisung definiert:

⟨*Clothoid module and import statements*⟩≡
```
1 module Clothoid;
```

Da die Entität IfcArbitraryTransitionSegment2D die Attribute StartPoint, SegmentLength und StartDirection von der Basisklasse IfcCurveSegment2D erbt und diese gegebenenfalls innerhalb des Konstruktors der IFC-PL-Klasse Clothoid benötigt werden, muss ein Weg gefunden werden, wie die IFC-PL-Klasse auf diese Attribute Zugriff erhält. Prinzipiell gibt es dafür unterschiedliche Lösungsvarianten.

Mit dem Begriff ererbte Daten sind im Folgenden die Daten der Entität IfcArbitraryTransitionSegment2D mit den Attributen StartPoint, SegmentLength und StartDirection gemeint. Tatsächlich besteht zwischen der Schnittstelle IArbitraryTransitionCurve und IfcArbitraryTransitionSegment2D keine Vererbungsbeziehung, jedoch könnten Realisierungen des Interfaces Zugriff auf die Daten einer Instanz der Entität IfcArbitraryTransitionSegment2D benötigen.

Variante A: Als eine der einfachsten Lösungsvarianten bietet sich an, dass man keinen Zugriff für ererbte Attribute anbietet. In diesem Fall müssten alle Attri-

[2]Da dieses Beispiel etwas umfangreicher ist, wird der Quelltext mithilfe der Literate Programming Technik beschrieben. Mehr Informationen zu Literate Programming finden sich im Anhang A.

bute, welche die IFC-PL-Klasse benötigt, mit ins Initialisierungs-Property-Set aufgenommen werden, das verwendet wird, um eine Instanz der entsprechenden IFC-PL-Klasse zu erzeugen. In diesem konkreten Fall benötigt der Konstruktor der IFC-PL-Klasse Clothoid Zugriff auf alle drei Attribute (StartPoint, SegmentLength und StartDirection) der Entität IfcArbitraryTransitionSegment2D. Diese Attribute müsste man entsprechend mit in das Initialisierungs-Property-Set aufnehmen. Dabei würde man dann aber diese Daten redundant speichern, einmal als Attributwerte in der Entität IfcArbitraryTransitionSegment2D, die diese von der Entität IfcCurveSegment2D erben, und zusätzlich würden die gleichen Werte im Initialisierungs-Property-Set gespeichert werden. Natürlich könnte man sich in diesem Fall über ein Redesign der Entität IfcCurveSegment2D Gedanken machen, das diese doppelte Datenhaltung vermeidet. Beispielsweise könnte man die Entität IfcCurveSegment2D so umgestalten, dass diese ganz ohne Parameter auskommt, bzw. die benötigten Parameter in einer Oberklasse auslagern, so dass diese nicht mehr doppelt vorkommen.

Variante B: Eine andere technische Lösung wäre, das Property-Set mit den ererbten Daten zu ergänzen, bevor es an den Konstruktor übergeben wird. Dabei würde das Poperty-Set zunächst nicht die Attribute StartPoint, SegmentLength und StartDirection enthalten, aber vor dem Aufruf des Konstruktors würde automatisiert ermittelt, welche ererbten Daten vorliegen, und diese in das Property-Set übernommen werden. Hierbei müsste klar definiert werden, wie und wann die Daten ins Property-Set automatisiert übernommen werden.

Variante C: Eine weitere Lösung des Problems besteht darin, dass der Konstruktor neben einem IfcPropertySet auch die ererbten Parameter erwartet. Der Konstruktor würde dann im Klothoidenbeispiel folgende Gestalt besitzen:

```
1 Clothoid(IfcCartesianPoint StartPoint,
2          IfcPlaneAngleMeasure StartDirection,
3          IfcPositiveLengthMeasure SegmentLength,
4          IfcPropertySet properties) {
5    // ...
6 }
```

Hierbei würde für die Signatur des Konstruktors folgende Regel gelten: Erst werden der Reihe nach alle vererbten Attribute als Parameter aufgelistet und als letztes Argument folgt das eigentliche Initialisierungs-Property-Set. Da keine direkte Vererbungsbeziehung zwischen der Schnittstelle IArbitraryTransitionCurve bzw. der IFC-PL-Klasse Clothoid und der Entität IfcArbitraryTransitionSegment2D besteht, ist es nicht ganz trivial, herauszufinden, welche Attribute in die Signatur für den Konstruktor aufgenommen werden müssen.

Für die Realisierung der IFC-PL-Umgebung wurde Variante *B* gewählt. Hierbei wird erwartet, dass vor Aufruf des Konstruktors der *IfcPropertySet*-Parameter entsprechend mit den ererbten Parametern erweitert wird. Sollte es keine ererbten Parameter geben, muss das Property-Set auch nicht erweitert werden. Die Erweiterung des Property-Sets mit ererbten Attributen muss beim Import der entsprechenden Daten erfolgen.

Der Konstruktor der Klasse Clothoid erhält als Initialisierungsparameter nur eine Variable vom Typ IfcPropertySet wie in Variante *B* beschrieben. Von diesem Konstruktor wird allerdings erwartet, dass dieser bereits um die ererbten Attribute angereichert wurde. Die Klasse IfcPropertySet selbst ist im IFC 4.1-Schema definiert. Um diese dem Modul bzw. der Klasse Clothoid bekannt zu machen, wird eine entsprechende Import-Anweisung verwendet:

⟨*Clothoid module and import statements*⟩+≡
```
1 import IFC4X1.exp;
```

Weiterhin benötigt die Implementierung die Definition der Schnittstelle IArbitraryTransitionCurve und einige mathematische Funktionen aus dem Modul Math.

⟨*Clothoid module and import statements*⟩+≡
```
1 import Math;
2 import ArbitraryTransitionCurve;
```

Das Interface IArbitraryTransitionCurve nutzt selbst das Hilfsinterface IVector2D. Dieses beschreibt einen 2D-Vektor, der den entsprechenden x- und y-Wert liefert. Die Klasse Vector2DImpl implementiert dieses Interface und ist wie folgt definiert:

⟨*IVector2D implementation*⟩≡
```
1  class Vector2DImpl : IVector2D {
2      public Vector2DImpl(const double x, const double y) {
3          this.x = x;
4          this.y = y;
5      }
6
7      public double X() const {
8          return x;
9      }
10
11     public double Y() const {
12         return y;
13     }
14
15     private double x;
16     private double y;
17 }
```

Die Klasse Clothoid stellt einen Konstruktor, öffentliche und private Methoden sowie Membervariablen bereit.

⟨*IArbitraryTransitionCurve implementation*⟩≡
```
1 class Clothoid : IArbitraryTransitionCurve {
2     ⟨Clothoid constructor 144⟩
3     ⟨Clothoid public methods 148⟩
4     ⟨Clothoid private methods 145⟩
5     ⟨Clothoid member variables 144⟩
6 }
```

Die Membervariablen der Klothoiden-Klasse speichern die Daten, die an den Konstruktor mittels eines IfcPropertySets übergeben werden, ab, im Detail die

Startposition (startPosition_), die Startrichtung (startDirection_), die Startkrümmung (startCurvature_), ein Attribut, das anzeigt, ob der Klothoidenbogen gegen oder im Uhrzeigersinn beginnt (counterClockwise_), den Klothoidenparameter (clothoidConstant_), die Angabe, ob die Krümmung vom Start- zum Endpunkt der Klothoide zu- oder abnimmt (entry_), und die Länge des Klothoidenbogens (length_).

⟨*Clothoid member variables*⟩≡

```
1 private Vector2d startPosition_;
2 private double   startDirection_;
3 private double   startCurvature_;
4 private bool     counterClockwise_;
5 private double   clothoidConstant_;
6 private bool     entry_;
7 private double   length_;
```

Basierend auf diesen Daten werden im Konstruktor zwei weitere Hilfsparameter berechnet und in Membervariablen vorgehalten:

⟨*Clothoid member variables*⟩+≡

```
1 private double   startL_;
2 private double   endL_;
```

Bei diesen Parametern handelt es sich um die Länge des Bogens vom Klothoidenanfang bis zum Start- bzw. Endpunkt des Klothoidenausschnitts, der entsprechend in die Trassierung eingepasst werden soll.

Initialisierung der Klasse Clothoid

Der Konstruktor liest das IfcPropertySet aus und initialisiert entsprechend alle Membervariablen:

⟨*Clothoid constructor*⟩≡

```
1  public Clothoid(IfcPropertySet ps) {
2      ⟨Get clothoid data from property set 145⟩
3
4      startL_ = clothoidConstant_ * clothoidConstant_ * startCurvature_;
5
6      if (entry_)
7          endL_ = startL_ + length_;
8      else
9          endL_ = max(startL_ - length_, 0.0);
10 }
```

Als Eingabeparameter erhält die Klothoiden-Klasse ein Property-Set. Die Klasse IfcPropertySet speichert im Attribut HasProperties vom Typ IfcProperty alle definierten Eigenschaftswerte ab. Diese werden ausgelesen und entsprechend in den Membervariablen abgespeichert. Die Klasse IfcProperty stellt zum Speichern verschiedene abgeleitete Klassen zur Verfügung. Im Fall der Klothoidenklasse wird hierbei auf die Klassen IfcPropertySingleValue und IfcPropertyListValue zurückgegriffen. Dabei findet ein Downcast vom Basisklassentyp IfcProperty zu den spezialisierten Datentypen statt.

⟨*Get clothoid data from property set*⟩≡

```
for(int i = 0; i < ps.HasProperties.count(); ++i) {
    if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
        IfcPropertySingleValue s =
          cast<IfcPropertySingleValue>(ps.HasProperties[i]);
        IfcIdentifier s_id = s.Name;

        ⟨Read IfcPropertySingleValue property clothoid data 145⟩
    }

    if(ps.HasProperties[i] instanceof IfcPropertyListValue) {
        IfcPropertyListValue s =
          cast<IfcPropertyListValue>(ps.HasProperties[i]);
        IfcIdentifier s_id = s.Name;

        if(s_id.getValue() == "StartPoint") {
            IfcLengthMeasure x = cast<IfcLengthMeasure>(s.ListValues[0]);
            IfcLengthMeasure y = cast<IfcLengthMeasure>(s.ListValues[1]);

            startPosition_ = new Vector2d(x.getValue(), y.getValue());
        }
    }
}
```

Der Programmcode zum Auslesen der IfcPropertySingleValue Eigenschaftswerte ist im Folgenden dargestellt.

⟨*Read IfcPropertySingleValue property clothoid data*⟩≡

```
readReal(s, "StartDirection", startDirection_);
readReal(s, "SegmentLength", length_);
readReal(s, "startCurvature", startCurvature_);
readReal(s, "clothoidConstant", clothoidConstant_);
readBool(s, "counterClockwise", counterClockwise_);
readBool(s, "entry", entry_);
```

Die Hilfsmethoden readReal und readBool überprüfen, ob der Name des Properties mit dem entsprechenden Attributnamen übereinstimmt, und initialisiert das Attribut gegebenenfalls.

⟨*Clothoid private methods*⟩≡

```
private void readReal(IfcPropertySingleValue ifcPSV, string name,
                      ref double outValue) {
    IfcIdentifier s_id = ifcPSV.Name;

    if(s_id.getValue() == name) {
        IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
        outValue = ifcReal.getValue();
    }
}

private void readBool(IfcPropertySingleValue ifcPSV, string name,
                      ref bool outValue) {
    IfcIdentifier s_id = ifcPSV.Name;
```

```
14
15      if(s_id.getValue() == name) {
16          IfcBoolean ifcBoolean = cast<IfcBoolean>(s.NominalValue);
17          outValue = ifcBoolean.getValue();
18      }
19  }
```

Berechnung der x-, y-Koordinaten

Die Klasse Clohtoid stellt eine Methode (computeX) zur Berechnung der x-Koordinate einer Klothoide in Abhängigkeit einer Klothoidenkonstante A und der Bogenlänge L bereit. Die Herleitung dieser Berechnung ist im Anhang B.3.1.1 zu finden.

⟨*Clothoid private methods*⟩+≡

```
1  private static double computeX(const double L, const double A) const {
2      double x = L;
3      const int iterations = 5;
4
5      for (int i = 1; i < iterations+1; i++) {
6          double sign = i % 2 == 0 ? 1 : -1;
7
8          double L_exponent = 5+(i-1)*4;
9          double A_exponent = i*4;
10         double factor = factorial(2*i) * pow(2.0, 2*i) * (5+(i-1)*4);
11         factor = factor + 3.0;
12         double debug = pow(A, A_exponent);
13         x += sign * pow(L, L_exponent) / (factor * debug);
14     }
15
16     return x;
17 }
```

Es werden nur die ersten fünf Glieder berechnet (const int iterations = 5;). Aufgrund der Fakultätsfunktion im Nenner der einzelnen Glieder, die mit zunehmender Gliederanzahl rasant ansteigt, wird der Gesamteinfluss aller Folgeglieder mit typischen Werten für A und L aus der Trassierungsplanung immer geringer und ist nach einigen wenigen Iterationen im vernachlässigbaren Bereich.

Für die Berechnung des y-Werts einer Klothoide wird ebenfalls eine Methode angeboten, die abhängig vom Längenwert und Klothoidenparameter die Koordinate berechnet.

⟨*Clothoid private methods*⟩+≡

```
1  private static double computeY(const double L, const double A) const {
2      double y = 0;
3      const int iterations = 5;
4
5      for (int i = 0; i < iterations; i++) {
6          double sign = i % 2 == 0 ? 1 : -1;
7
8          double L_exponent = 3+i*4;
```

```
9           double A_exponent = 2+i*4;
10          double factor = factorial(2*i+1) * pow(2.0, i*2) * 2 *
11              (A_exponent + 1);
12
13          y += sign * pow(L, L_exponent) / (factor * pow(A, A_exponent));
14      }
15
16      return y;
17  }
```

Die beiden Methoden computeX und computeY berechnen die Koordinaten der Klothoide in ihrem eigenen lokalen Koordinatensystem. Abbildung 8.7 zeigt eine Klothoide in ihrem lokalen Koordinatensystem. Die Klothoide hat ihren Startpunkt in diesem Koordinatensystem im Ursprung. Typischerweise wird nur ein Teilbereich der Klothoide verwendet. Dieser Teilbereich ist mit einem Start- und Endpunkt markiert.

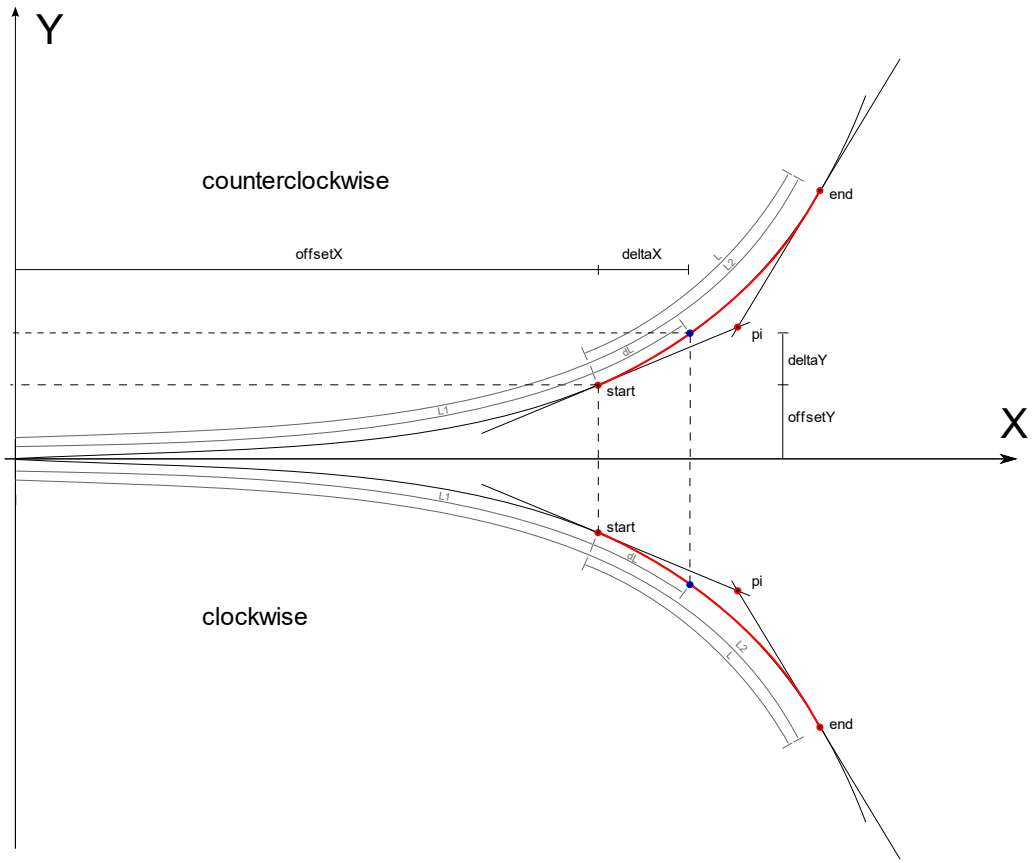


Abbildung 8.7: Klothoide in ihrem lokalen Koordinatensystem

Beim Einpassen der Klothoide in eine bestimmte Trassierungssituation muss die Klothoide noch von ihrem lokalen Koordinatensystem in das globale Koordinatensystem der Trasse überführt werden. Die lokalen Koordinaten werden durch die Methode computeLocalPosition berechnet.

⟨*Clothoid private methods*⟩+≡

```
1  private Vector2d computeLocalPosition(const double L) const {
2      Vector2d localPosition = new Vector2d();
3
4      localPosition.x() = computeX(L, clothoidConstant_);
5      localPosition.y() = computeY(L, clothoidConstant_);
6
7      return localPosition;
8  }
```

Um die globalen Koordinaten berechnen zu können, werden die Hilfsmethoden computeTau und isEntry eingeführt. Die Methode computeTau berechnet den Winkel zwischen der Tangente am Endpunkt und der Tangente am Startpunkt der Klothoide mit der Länge L und dem Klothoidenparameter A.

```
⟨Clothoid private methods⟩+≡
1  private static double computeTau(const double L, const double A) const {
2      return L*L / (2 * A*A);
3  }
```

Das Attribut entry_ der Klothodien-Klasse definiert, ob die Krümmungsänderung innerhalb des Spiralbogens vom Startpunkt aus zum Endpunkt eine zunehmende Krümmung aufweist (in diesem Fall ist der Wert von entry_ gleich true), oder ob der Bogen der Klothoide eine abnehmende Krümmung aufweist (entry_ gleich false). Diese Definition geht zurück auf eine Entwicklung aus dem buildingSMART Projekt IFC Alignment 1.0 (buildingSMART, 2018d). Die Methode isEntry liefert den entsprechenden Wert.

```
⟨Clothoid private methods⟩+≡
1  private bool isEntry() const {
2      return entry_;
3  }
```

Die globale Position wird durch die Methode getPostion berechnet. Dabei wird neben den vorher beschriebenen Hilfsfunktionen noch eine weitere Funktion (createRotationMatrix) verwendet. Diese ist im Modul Math definiert. Sie erzeugt eine Rotationsmatrix, die die Koordinaten entsprechend in das verdrehte globale System transformiert.

```
⟨Clothoid public methods⟩≡
1  public IVector2D getPostion(const double lerpParameter) const override {
2      double L = startL_ + (endL_ - startL_) * lerpParameter;
3
4      Vector2d localPosition = computeLocalPosition(L);
5
6      double angle = computeTau(startL_, clothoidConstant_);
7      Vector2d localOffset = localPosition - computeLocalPosition(startL_);
8
9      if (!isEntry()) {
10         angle *= -1;
11         localOffset.x() *= -1;
12     }
13
```

```
14      if (!counterClockwise_) {
15          angle *= -1;
16      }
17
18      Vector2d position = startPosition_ +
          createRotationMatrix(startDirection_ - angle) * localOffset;
19
20      return new Vector2DImpl(position.x(), position.y());
21  }
```

Abbildung 8.8 zeigt exemplarisch die Drehung eines Klothoidenastes, die durch die Methode getPostion durchgeführt wird.

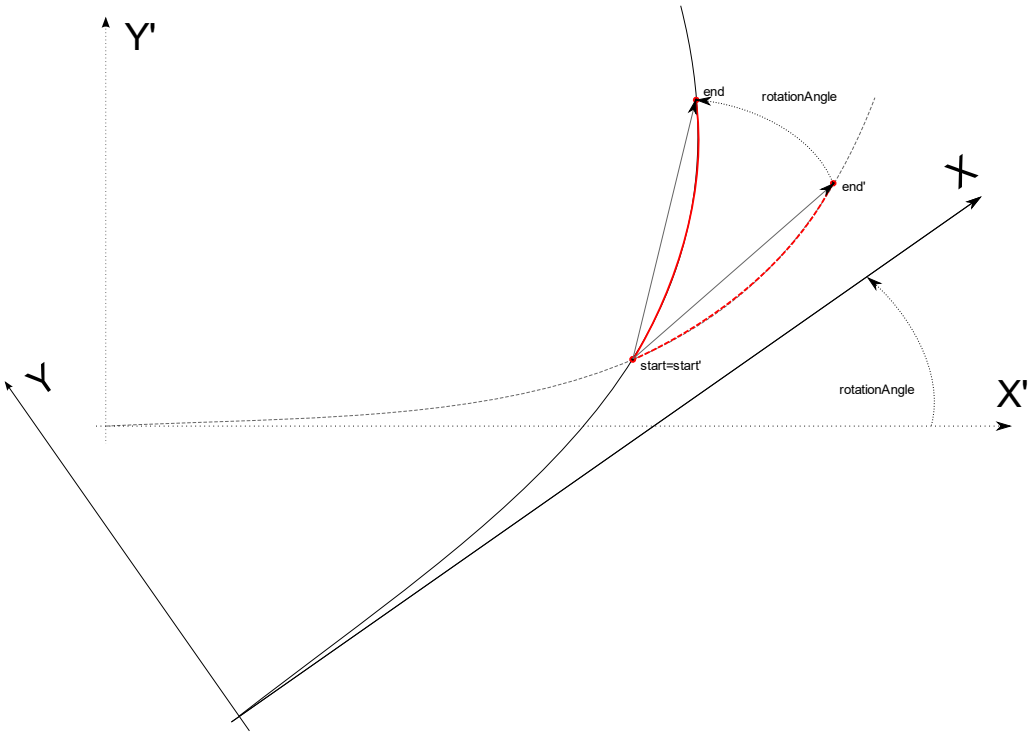


Abbildung 8.8: Klothoide im globalen Koordinatensystem

Berechnung der Krümmung und Länge

Die Krümmung der Klothoide wird durch die Methode getCurvature berechnet. Dabei wird der lineare Interpolationsparameter in einen Längenwert umgerechnet und daraus die Krümmung berechnet.

⟨*Clothoid public methods*⟩+≡

```
1  public double getCurvature(const double lerpParameter) const override {
2      double L = startL_ + (endL_ - startL_) * lerpParameter;
3      return L / (A * A);
4  }
```

Die Länge der Klothoide kann durch die Differenz zwischen der Länge zum Start- und Endpunkt des Klothoidenausschnittes berechnet werden:

⟨*Clothoid public methods*⟩+≡

```
1  public double getLength() const override {
2      return abs(endL_ - startL_);
3  }
```

Weitere Implementierungsdetails

Die Methode getParamters der Klasse Clothoid liefert als Rückgabewert ein IfcPropertySet, das die frei definierten Parameter (startCurvature, counterClockwise, clothoidConstant und entry) enthält.

⟨*Clothoid public methods*⟩+≡

```
1  public IfcPropertySet getParamters() {
2      IfcPropertySingleValue StartCurvature = new IfcPropertySingleValue();
3      StartCurvature.Name = new IfcIdentifier("startCurvature");
4      StartCurvature.NominalValue = new IfcReal(startCurvature_);
5
6      IfcPropertySingleValue CounterClockwise = new
         IfcPropertySingleValue();
7      CounterClockwise.Name = new IfcIdentifier("counterClockwise");
8      CounterClockwise.NominalValue = new IfcBoolean(counterClockwise_);
9
10     IfcPropertySingleValue ClothoidConstant = new
         IfcPropertySingleValue();
11     ClothoidConstant.Name = new IfcIdentifier("clothoidConstant");
12     ClothoidConstant.NominalValue = new IfcReal(clothoidConstant_);
13
14     IfcPropertySingleValue Entry = new IfcPropertySingleValue();
15     Entry.Name = new IfcIdentifier("entry");
16     Entry.NominalValue = new IfcBoolean(entry_);
17
18     IfcPropertySet ps = new IfcPropertySet();
19     ps.Name = new IfcLabel("Parameters");
20     ps.HasProperties.add(StartCurvature);
21     ps.HasProperties.add(ClothoidConstant);
22     ps.HasProperties.add(CounterClockwise);
23     ps.HasProperties.add(Entry);
24
25     return ps;
26 }
```

Der IFC-PL-Quelltext der Klasse Clothoid befindet sich zusammenhängend in einem Stück im Anhang B.3.1.2.

8.1.3.4 Datenaustausch: Import

Beim Export einer Klothoide aus einer Applikation wird wie gewöhnlich beim konventionellen Ansatz eine STEP-P21-Datei geschrieben. Diese STEP-Datei enthält eine Referenz auf das Klothoidenprogramm. Der Quelltext des Klothoiden-

programms bzw. der Klothoidenklasse muss zusätzlich exportiert werden. Dieser wird in einer Datei mit dem Namen Clohtoid.ifcpl gespeichert. Beide Dateien können dann zu einer Applikation weitergereicht werden, die diese Daten importieren möchte.

Im IFC-PL-Ansatz wird der Host-Language-Interface-Generator nun genutzt, um einem Programm die Schnittstellendefinition dieser Implementierung (der Klasse Clohtoid) bekannt zu machen.

Der importierenden Applikation muss die Schnittstelle, die wir für Übergangskurven festgelegt haben (IArbitraryTransitionCurve), bekannt sein. Im konventionellen Ansatz verhält sich dies ähnlich. Dort muss dem importierenden Programm auch bekannt gemacht werden, welche Übergangskurven es gibt. Zudem muss festgelegt sein, welche Parameter diese Übergangskurven besitzen, und anschließend auf Basis der Dokumentation eine Implementierung entwickelt werden, was die eingangs beschriebenen Nachteile mit sich bringt.

Bei der Umsetzung mittels IFC-PL muss dem importierenden Programm nur die Schnittstelle bekannt gemacht werden. Dieser Schritt kann manuell durchgeführt oder durch den Host-Language-Interface-Generator unterstützt werden. Dieser enthält als Eingabe die Schnittstellendefinition (*IArbitraryTransitionCurve.ifcpl*) und erzeugt daraus für die Zielsprache der Applikation eine entsprechende Schnittstelle.

Nehmen wir an, unsere Zielapplikation ist in der Sprache C++ entwickelt. In diesem Fall erzeugt der Host-Language-Interface-Generator aus der Schnittstellenbeschreibung folgendes Quellcodefragment für die Zielsprache C++:

ArbitraryTransitionCurve.h:

```
1  #include <IFC4X1.h>
2  #include <memory>
3
4  class IVector2d {
5  public:
6      virtual double getX() const = 0;
7      virtual double getY() const = 0;
8  };
9
10 class IArbitraryTransitionCurve {
11 public:
12     virtual double getLength() const = 0;
13     virtual IVector2d getPostion(const double lerpParameter) const = 0;
14     virtual double getCurvature(const double lerpParameter) const = 0;
15 };
16
17 std::shared_ptr<IArbitraryTransitionCurve>
18 createArbitraryTransitionCurve(
19     const char* className,
20     IfcPropertySet* ps
21 );
```

Die Klasse IfcPropertySet wird in der Headerdatei IFC4x1.h definiert. Diese wird vom Host-Language-Interface-Generator durch einen Early-Binding-Generator auf Basis des verwendeten Schemas (in diesem Fall das IFC 4.1 Schema, erweitert durch die Entität IfcArbitraryTransitionSegment2D) generiert und enthält alle IFC-Entitäten als IFC-Klassen.

Die Implementierung der createArbitraryTransitionCurve-Funktion befindet sich in der Datei ArbitraryTransitionCurve.cpp, die ebenfalls vom Host-Language-Interface-Generator generiert wird. Diese bindet die Headerdatei IfcPL.h ein, die eine Klasse IFCPLEnvironment definiert, welche Teil der IFC-PL-Laufzeit-Umgebung ist. Diese Klasse ermöglicht es, zur Laufzeit Instanzen von IFC-PL-Klassen zu erzeugen, die auf Basis der IFC-PL definiert wurden, wie dies der Fall für die Klasse Clothoid ist. Die Klasse IFCPLEnvironment sowie die Header-Datei IfcPL.h sind Teil der IFC-PL-Umgebung. Folgendes Listing zeigt die vom Host-Language-Interface-Generator generierte Datei ArbitraryTransitionCurve.cpp.

ArbitraryTransitionCurve.cpp:

```
1  #include "ArbitraryTransitionCurve.h"
2  #include "IfcPL.h"
3
4  std::shared_ptr<IArbitraryTransitionCurve>
5  createArbitraryTransitionCurve(const char* className, IfcPropertySet* ps)
6     return std::shared_ptr<IArbitraryTransitionCurve>(
7        (IArbitraryTransitionCurve*)
8           IfcPLRuntimeEnvironment::create(className,ps)
9     );
10 }
```

Die Host-Applikation kann nun zur Laufzeit mittels der Funktion createArbitraryTransitionCurve beliebige Instanzen von Übergangskurven erzeugen, die mit der IFC-PL programmiert wurden.

Dabei müssen, wie in der vorherigen Sektion (Export) diskutiert, bei der Initialisierung der Property-Sets diese erst mit ererbten Attributen angereichert werden. Dies hat ebenfalls durch die Zielapplikation zu erfolgen[3]. Im Falle von beliebigen Übergangskurven erbt die Klasse IfcArbitraryTransitionSegment2D die Attribute StartPoint, StartDirection und SegmentLength, die jeweils vom Typ IfcCartesianPoint, IfcPlaneAngleMeasure und IfcPositiveLengthMeasure sind. Problematisch an diesem Ansatz ist, dass die IfcPropertySet-Klasse nicht alle Typen unterstützt. Beispielsweise kann in einem Property-Set kein IfcCartesianPoint gespeichert werden. Um dieses Problem zu umgehen, bieten sich unterschiedliche Lösungsvarianten an.

Zum einen könnte man das EXPRESS-Schema entsprechend anpassen, so dass dieses auch die benötigten Typen unterstützt. Diese Variante bedeutet allerdings, das bestehende EXPRESS-Schema weiter modifizieren zu müssen. Die Modifika-

[3]Theoretisch könnte der Host-Language-Interface-Generator hierfür automatisiert auch eine Hilfsfunktion zur Verfügung stellen

tionen sind prinzipiell einfach durchzuführen, jedoch widerspricht diese Modifikation der bestehenden Umsetzung der Architektur von Property-Sets im IFC-Standard.

Eine andere Lösungsalternative ist, ein Boxing bzw. Unboxing zu definieren, das festlegt, wie Datentypen bestimmter Arten in Property-Sets verpackt und wieder entpackt werden können. Diese Variante wurde für die Umsetzung gewählt. Daten vom Typ IfcCartesianPoint werden demnach mittels des Datentyps IfcPropertyListValue abgebildet. Die lesende Applikation kann anhand des Namens des Attributs (StartPoint) erkennen, dass es sich hierbei um ein Attribut vom Typ IfcCartesianPoint handelt, was natürlich Kontextwissen erfordert, jedoch ein gangbarer Weg ist.

Beispielhaft wird dies im Folgenden für die Zielsprache C++ gezeigt:

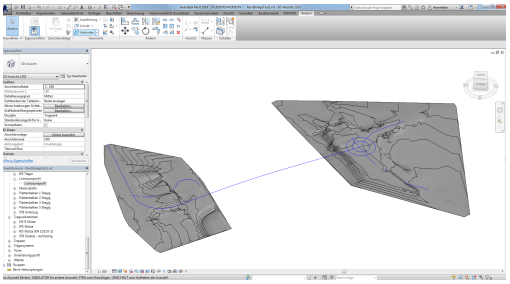
```
1  // 1. Box derived parameters
2  // IfcCartesianPoint is not supported by IfcPropertySet
3  std::shared_ptr<IfcPropertyListValue> StartPoint = std::make_shared<IfcPropertyListValue>(id++);
4  StartPoint->m_ListValues.push_back(x);
5  StartPoint->m_ListValues.push_back(y);
6
7  std::shared_ptr<IfcPropertySingleValue> StartDirection = std::make_shared<IfcPropertySingleValue>(id++);
8  StartDirection->m_Name = std::make_shared<IfcIdentifier>(IfcPLString("StartDirection"));
9  StartDirection->m_NominalValue = std::make_shared<IfcPlaneAngleMeasure>(5.070000);
10
11 std::shared_ptr<IfcPropertySingleValue> SegmentLength = std::make_shared<IfcPropertySingleValue>(id++);
12 SegmentLength->m_Name = std::make_shared<IfcIdentifier>(IfcPLString("SegmentLength"));
13 SegmentLength->m_NominalValue = std::make_shared<IfcPositiveLengthMeasure>(12.430000);
14
15 ...
16
17 // 2. Load property set
18 IfcPropertySet ps = ...
19
20 // 3. Extend property set
21 ps.m_HasProperties.push_back(StartPoint);
22 ps.m_HasProperties.push_back(StartDirection);
23 ps.m_HasProperties.push_back(SegmentLength);
```

Um diesen Anwendungsfall zu testen, wurde dieser, wie in diesem Abschnitt beschrieben, umgesetzt und in verschiedenen Softwareanwendungen prototypisch integriert. Zu den getesteten Anwendungen gehören Autodesk Revit, Siemens NX, Dynamo und die TUM Open Infra Platform (siehe Abbildung 8.9).

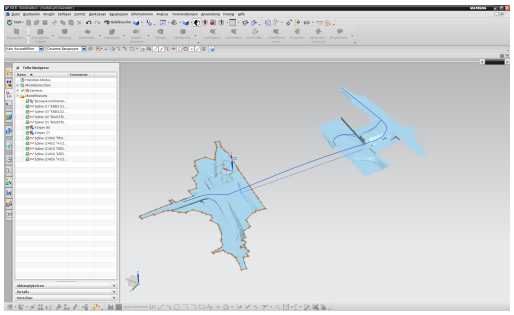
Verändert man die getPosition-Methode derart, dass diese, anstatt eine Klothoide zu berechnen (Abbildung 8.10a), einfach eine lineare Interpolation zwischen dem Start- und Endpunkt der Klothoide durchführt (Abbildung 8.10b), so wird auch im verarbeitenden Programm einfach eine Gerade anstatt einer Klothoide angezeigt.

Die Umsetzung auf Basis der IFC-PL führt zu dem Vorteil, dass nicht in jeder Applikation die Klothoide neu implementiert werden muss. Weiter bietet die Umsetzung den Vorteil, dass auch andere in IFC-PL formulierte Übergangskurven ohne weitere Änderungen an einer Fachapplikation automatisch unterstützt werden.

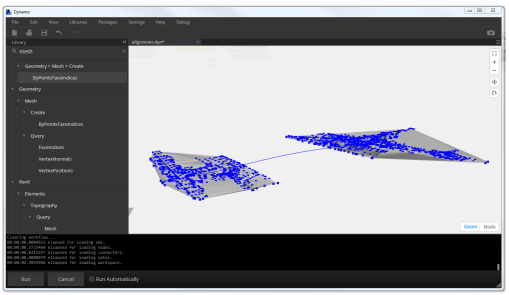
Autodesk Revit



Siemens NX



Dynamo



TUM Open Infra Platform

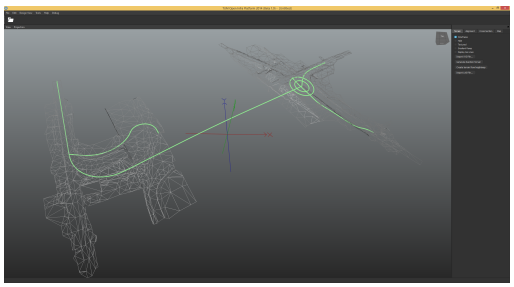
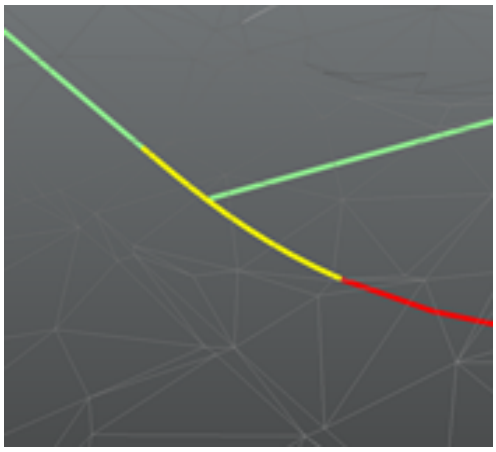
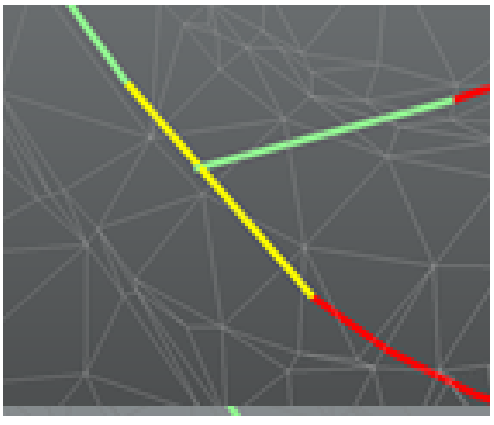


Abbildung 8.9: Datenaustausch von beliebigen Übergangskurven zwischen verschiedenen Applikationen: (a) Autodesk Revit, (b) Siemens NX, (c) Dynamo, (d) TUM Open Infra Platform



(a) Unmodifizierte getPosition-Methode



(b) getPosition-Methode führt lineare Interpolation zwischen Start- und Endpunkt durch.

Abbildung 8.10: Test mit modifizierter getPosition-Methode

8.2 Parametrische Geometriebeschreibungen von Volumenkörpern

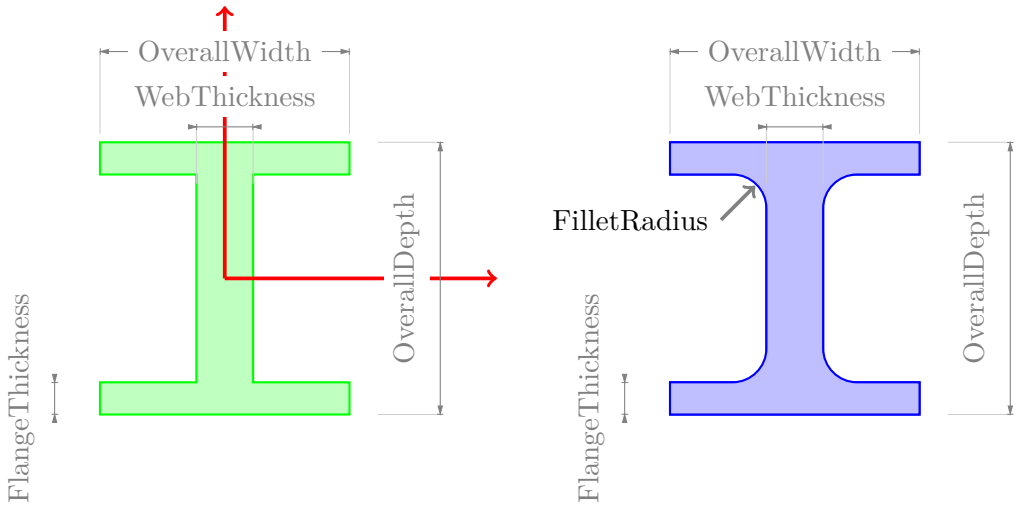
8.2.1 IFC-Geometrie-Daten

IFC 4.1 bietet vielfältige Möglichkeiten, zwei- und dreidimensionale Geometriedaten zu repräsentieren. Zur Beschreibung dreidimensionaler Geometrie werden verschiedene Ansätze unterstützt, im Wesentlichen Flächen- und Volumenmodelle (Amann *et al.*, 2015).

Volumenmodelle besitzen eine geschlossene Hülle. Volumenkörper können direkt oder indirekt modelliert werden (Bungartz *et al.*, 2013). Eine direkte Modellierungsmethode für Volumenmodelle ist Constructive Solid Geometry (CSG). CSG basiert auf einfachen definierten Grundkörpern wie z. B. Quader (IfcBlock), Pyramide (IfcRectangularPyramid) oder Zylinder (IfcRightCircularCylinder), die mittels boolescher Operationen miteinander vereinigt, geschnitten oder voneinander abgezogen werden können. Bei der indirekten Modellierung wird ein Volumenkörper durch die Beschreibung seiner Hüllflächen beschrieben. Typischerweise wird hier ein Boundary Representation Ansatz verwendet, der die umgrenzenden Flächen mittels Eckpunkten, Kanten und sich daraus ergebenden Flächen definiert. Die Flächen können eben oder gekrümmt sein. Ebene Flächen werden mit Polygonen beschrieben und gekrümmte Oberflächen können mithilfe von Non-Uniform Rational B-Splines (NURBS) beschrieben werden, die in IFC 4.1 eingeführt worden sind. Weiter ist es möglich, Objekte, die mit einer Boundary Representation beschrieben sind, gegen eine Schnittebene zu schneiden. Die Schnittebene definiert dabei einen positiven und einen negativen Halbraum. Mit ihr kann dadurch ein Teil eines Objekts, abhängig davon, ob er im positiven oder negativen Halbraum liegt, entsprechend abgetrennt werden. Als weitere Besonderheit können Opening-Elemente definiert werden, die z. B. genutzt werden können, um Durchbrüche in Wänden für Türen oder Fenster zu definieren. IFC 4.1 unterstützt ebenfalls Extrusionskörper. Dabei wird ein 2D-Profil entlang einer Linie oder Kurve parallel verschoben und dadurch indirekt ein 3D-Volumenkörper beschrieben. Diese Extrusionskörper können dann ebenfalls wieder als Operanden in einer CSG-Operation verwendet werden. Das IFC-Datenmodell definiert auch einige sogenannte parametrische Profildefinitionen (IfcParameterizedProfileDef). Insgesamt besitzt IFC 4.1 elf unterschiedliche Varianten dieser Profiledefinitionen, die im Folgendem aufgelistet sind:

- IfcAsymmetricIShapeProfileDef
- IfcCircleProfileDef
- IfcCShapeProfileDef
- IfcEllipseProfileDef
- IfcIShapeProfileDef
- IfcLShapeProfileDef
- IfcRectangleProfileDef
- IfcTrapeziumProfileDef
- IfcTShapeProfileDef
- IfcUShapeProfileDef
- IfcZShapeProfileDef

Ein Beispiel für eine parametrische Profildefinition (IfcIShapeProfileDef) ist in Abbildung 8.11 dargestellt.



(a) Doppel-T-Profil ohne Ausrundungen **(b)** Doppel-T-Profil mit Ausrundungen

Abbildung 8.11: Der Doppel-T-Träger (IfcIShapeProfileDef) wird anhand der Attribute OverallWidth, OverallDepth, WebThickness, FlangeThickness, FilletRadius, FlangeEdgeRadius und FlangeSlope definiert. Die letzten drei Parameter sind optional und beschreiben die Rundungsradien an den Übergängen des Trägers.

Eine Applikation, die volle Unterstützung für alle Geometrievarianten der IFC 4.1 bieten möchte, muss mit all den verschiedenen Beschreibungsmöglichkeiten zurechtkommen. Dies ist mit einem erheblichen Aufwand auf Seiten der Softwareentwicklung verbunden. Model View Definitions (MVDs) mildern diesen Umstand für Softwareentwickler etwas. Mit einer MVD wird das IFC-Schema auf eine Teilmenge von Entitäten und Attributen beschränkt, die für ein bestimmtes Datenaustauschszenario benötigt bzw. unterstützt werden. Gleichzeit müssen dadurch alle Entitäten und Attribute, die außerhalb dieser Teilmenge liegen, nicht mehr unterstützt werden, was den Softwareentwicklungsaufwand drastisch reduzieren kann. Beispielsweise könnte eine MVD nur auf Trassierungsdaten und einfache dreiecksbasierte Oberflächenbeschreibungen beschränkt werden. Durch das Verbieten von parametrischen Profilen oder höherwertigen Geometriebeschreibungen wie Extrusion verliert das Datenmodell jedoch auch etwas von seiner Nützlichkeit. Benutzer sind evtl. daran interessiert, nicht nur eine ausgewertete Dreiecksbeschreibung zu erhalten, sondern möchten evtl. auch in der Lage sein zu sehen, auf welcher Grundlage die Dreiecksbeschreibung entstanden ist, und gegebenenfalls daran auch Änderungen vornehmen. Konkret könnte das Dreiecksnetz auf Basis einer Profilextrusion entstanden sein. Bei einer reinen Dreiecksgeometrie, die keinen Bezug mehr zur ursprünglichen Extrusion besitzt, wird es dann relativ schwierig, im Nachhinein eine Änderung am Extrusionsprofil vorzunehmen. Oft wird von Anwendern ein Design-To-Design-Datenaustausch gewünscht. Daher sind Beschränkungen des Datenmodells durch MVDs nur bedingt hilfreich. Darüber hinaus ist es auch denkbar, dass in Zukunft, um einen Design-To-Design-Transfer zu unterstützen, auch die Komplexität der Geometriebeschreibungen zunimmt. Es bleibt festzuhalten, dass sich ein gewisser Programmieraufwand nicht

verhindern lässt, wenn höherwertige geometrische Konstrukte der IFC 4.1 unterstützt werden sollen. Dieser Aufwand muss von jedem Softwareanbieter geleistet werden, der das Datenmodell unterstützen möchte. Dabei müssen Entwickler die Dokumentation richtig interpretieren und es muss sichergestellt werden, dass keine Fehler bei der Umsetzung von Import- bzw. Exportschnittstellen gemacht werden.

8.2.2 Flexible Unterstützung von parametrischen Profildefinitionen

Kommen wir zurück auf das Beispiel der parametrischen Profildefinitionen der IFC. Der verfolgte Ansatz in der IFC ist sehr starr, da er auf elf vorgefertigte parametrische Profildefinitionen festgelegt ist. Um beispielsweise eine neue parametrische Profildefinition in das IFC-Datenmodell einzuführen, muss diese zuvor den Standardisierungsprozess durchlaufen. Nach der Standardisierung muss der Standard von den Softwarehäusern umgesetzt und entsprechend korrekt implementiert werden. Dieser Prozess kann mehrere Jahre in Anspruch nehmen. Auf Basis der IFC-PL kann die Einführung von neuen parametrischen Profiltypen mit beliebigen Parametern ohne umständliche Standardisierung und softwaretechnische Umsetzung durch entsprechende Fachapplikationen realisiert werden.

8.2.2.1 Definition einer Schnittstelle

In Abbildung 8.11 ist ein UML-Klassendiagramm dargestellt, das die Schnittstelle für eine flexible Definition von parametrischen Profilen zeigt. Die Schnittstelle definiert zwei Methoden: Die Methode getBasicProfileDescription liefert die Profilbeschreibung auf Basis der bereits bekannten IFC-Klasse IfcProfileDef. Entsprechend liefert die Methode getParameters ein Property-Set, das alle Daten, auf deren Basis das Profil definiert ist, beschreibt.

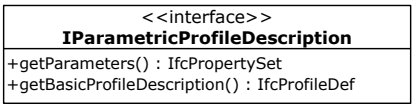


Abbildung 8.12: UML-Klassendiagramm, das die Schnittstelle für eine flexible Definiton von parametrischen Profilen zeigt

Die Schnittstelle wird formal mithilfe der IFC-PL (*IParametricProfileDescription.ifcpl*) wie folgt beschrieben:

```
1 module ParametricProfileDescription;
2
3 import IFC4X1.exp;
4
5 interface IParametricProfileDescription {
6    IfcPropertySet getParameters();
7    IfcProfileDef getBasicProfileDescription();
8 }
```

8.2.2.2 EXPRESS-Schemaerweiterung

Zur Unterstützung dieser flexiblen Beschreibung durch das IFC 4.1-Modell muss das EXPRESS-Schema entsprechend modifiziert werden. Abbildung 8.13 zeigt, wie das Datenmodell geändert werden muss, um den vorgestellten Vorschlag zu unterstützen. Es werden zwei neue Klassen eingeführt (IfcParametricIFCPLProfileDescription und IfcPLInterfaceRealizationConstructionDescription). Die Klasse IfcProfileDef wird unverändert aus dem IFC 4.1-Datenmodell übernommen. Sie dient als Basisklasse für die Klasse IfcParametricIFCPLProfileDescription. Diese wiederum besitzt ein Attribut mit dem Namen ConstructionDescription, das vom Typ IfcPLInterfaceRealizationConstructionDescription ist und entsprechend den Klassennamen der IFC-PL-Klasse enthält, die das Interface IParametricProfileDescription realisiert, und ein Property-Set, das dem Konstruktor der IFC-PL-Klasse zur Initialisierung bereitgestellt wird.

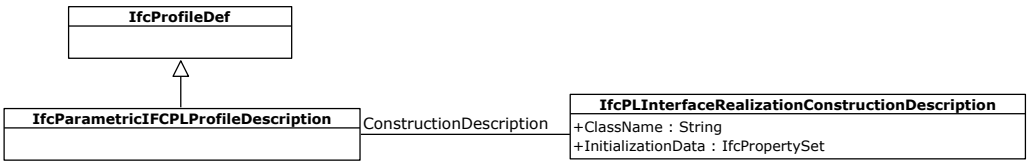


Abbildung 8.13: UML-Klassendiagramm, das die Erweiterung des IFC 4.1-EXPRESS-Schemas zeigt

8.2.2.3 Eine IFC-PL-Klasse für benutzerdefinierte Profile

Als benutzerdefiniertes Profil soll hier ein Doppel-T-Trägerprofil implementiert werden. Dieses existiert zwar bereits im IFC 4.1-Modell (IfcIShapeProfileDef), jedoch soll hier nur die prinzipielle Vorgehensweise veranschaulicht werden, wie beliebige Profiltypen auf Basis der IFC-PL unterstützt werden können. Dabei wird nur die Variante ohne Ausrundungsparameter realisiert, d. h. ohne Unterstützung für die optionalen Attribute FilletRadius, FlangeEdgeRadius und FlangeSlope, die in der Klasse IfcIShapeProfileDef vorgesehen sind (siehe Abbildung 8.11 links).

Das Grundgerüst für das IFC-PL-Programm wird im folgenden Codefragment dargestellt:

⟨*IShapeProfile.ifcpl*⟩≡

```
1  module IShapeProfile;
2
3  import ParametricProfileDescription;
4  import IFC4X1.exp;
5
6  class IShapeProfile : IParametricProfileDescription {
7      ⟨IShapeProfile constructor 159⟩
8
9      ⟨IShapeProfile readReal 159⟩
10     ⟨IShapeProfile getParameters 160⟩
11     ⟨IShapeProfile getBasicProfileDescription 160⟩
12     ⟨IShapeProfile createPoint 161⟩
13
14     ⟨IShapeProfile member variables 159⟩
```

```
15 }
```

Die Klasse erhält als Membervariablen alle Attribute (OverallWidth, OverallDepth, WebThickness und FlangeThickness), die benötigt werden, um den Doppel-T-Träger ohne Ausrundungen zu beschreiben.

⟨*IShapeProfile member variables*⟩≡

```
1 private double OverallWidth_;
2 private double OverallDepth_;
3 private double WebThickness_;
4 private double FlangeThickness_;
```

Der Konstruktor erwartet ein Property-Set, das die initialen Startwerte für diese Membervariablen enthält. Dieses wird beim Export einer Trassierung, die die IShapeProfile-Klasse verwendet, entsprechend mit in die STEP-Datei gespeichert. Ein Beispielauszug aus einer solchen STEP-Datei ist im folgenden Fragment zu sehen:

```
1 #100=IFCPROPERTYSET($,$,'Initialization Parameters',$,(#101,#102,#103,#104));
2 #101=IFCPROPERTYSINGLEVALUE('OverallWidth',$,100.,$);
3 #102=IFCPROPERTYSINGLEVALUE('OverallDepth',$,.220.,$);
4 #103=IFCPROPERTYSINGLEVALUE('WebThickness',$,5.9,$);
5 #104=IFCPROPERTYSINGLEVALUE('FlangeThickness',$,9.2,$);
```

Im Konstruktor der IShapeProfile-Klasse werden die Werte des Property-Sets entsprechend ausgelesen. Das Property-Set enthält die entsprechenden Properties im Attribut HasProperties, das eine Menge von Properties repräsentiert. Folgendes Listing zeigt, wie schrittweise über alle Elemente dieser Menge iteriert wird:

⟨*IShapeProfile constructor*⟩≡

```
1  IShapeProfile(IfcPropertySet ps) {
2     for(int i = 0; i < ps.HasProperties.count(); ++i) {
3        if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
4           IfcPropertySingleValue s =
              cast<IfcPropertySingleValue>(ps.HasProperties[i]);
5           IfcIdentifier s_id = s.Name;
6
7           readReal(s, "OverallWidth", OverallWidth_);
8           readReal(s, "OverallDepth", OverallDepth_);
9           readReal(s, "WebThickness", WebThickness_);
10          readReal(s, "FlangeThickness", FlangeThickness_);
11       }
12    }
13 }
```

Das Auslesen der Werte der verschiedenen Attribute erfolgt über die Methode readReal, die im Folgenden definiert ist:

⟨*IShapeProfile readReal*⟩≡

```
1 private void readReal(IfcPropertySingleValue ifcPSV, string name,
2                       ref double outValue) {
3    IfcIdentifier s_id = ifcPSV.Name;
4
```

```
5     if(s_id.getValue() == name) {
6         IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
7         outValue = ifcReal.getValue();
8     }
9  }
```

Die Methode getParameters liefert alle Parameter, die die Klasse intern verwendet, als Property-Set an die aufrufende Umgebung zurück.

⟨*IShapeProfile getParameters*⟩≡

```
1  public IfcPropertySet getParameters() {
2     IfcPropertySingleValue OverallWidth = new IfcPropertySingleValue();
3     OverallWidth.Name = new IfcIdentifier("OverallWidth");
4     OverallWidth.NominalValue = new IfcReal(OverallWidth_);
5
6     IfcPropertySingleValue OverallDepth = new IfcPropertySingleValue();
7     OverallDepth.Name = new IfcIdentifier("OverallDepth");
8     OverallDepth.NominalValue = new IfcReal(OverallDepth_);
9
10    IfcPropertySingleValue WebThickness = new IfcPropertySingleValue();
11    WebThickness.Name = new IfcIdentifier("WebThickness");
12    WebThickness.NominalValue = new IfcReal(WebThickness_);
13
14    IfcPropertySingleValue FlangeThickness = new
        IfcPropertySingleValue();
15    FlangeThickness.Name = new IfcIdentifier("FlangeThickness");
16    FlangeThickness.NominalValue = new IfcReal(FlangeThickness_);
17
18    IfcPropertySet ps = new IfcPropertySet();
19    ps.Name = new IfcLabel("Parameters");
20    ps.HasProperties.add(OverallWidth);
21    ps.HasProperties.add(OverallDepth);
22    ps.HasProperties.add(WebThickness);
23    ps.HasProperties.add(FlangeThickness);
24
25    return ps;
26 }
```

Durch diese Methode ist es einer Applikation möglich, die Parameter eines Profils beispielsweise in einem User-Interface grafisch zu repräsentieren (siehe Abbildung 8.14)

Die Methode getBasicProfileDescription muss das Profil eines Doppel-T-Trägers auf Basis der Klasse IfcProfileDef zurückgeben. Dabei nutzt die Methode die Unterklasse IfcArbitraryClosedProfileDef, die von der Klasse IfcProfileDef erbt. Die Klasse IfcArbitraryClosedProfileDef besitzt ein Attribut OuterCurve vom Typ IfcCurve, das die äußere Kontur eines Profils beschreibt. Eine Unterklasse von IfcCurve ist die Klasse IfcPolyline. Diese wird in der getBasicProfileDescription-Methode genutzt, um das Profil des Doppel-T-Trägers zu beschreiben. Folgendes IFC-PL-Fragment zeigt die beschriebene Vorgehensweise:

⟨*IShapeProfile getBasicProfileDescription*⟩≡

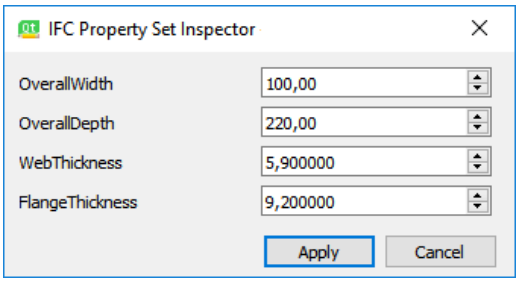


Abbildung 8.14: Beispielhafte Visualisierung eines Property-Sets in einer Fachapplikation

```
1  public IfcProfileDef getBasicProfileDescription() {
2      IfcPolyline polyline = new IfcPolyline();
3
4      ⟨IShapeProfile compute points 162⟩
5
6      IfcArbitraryClosedProfileDef profile = new
         IfcArbitraryClosedProfileDef();
7
8      profile.ProfileType = new
         IfcProfileTypeEnum(IfcProfileTypeEnum.AREA);
9      profile.OuterCurve = polyline;
10
11     return profile;
12 }
```

Abbildung 8.15 zeigt das zu erzeugende Doppel-T-Profil. Jeder Eckpunkt des Profils wurde mit einen Buchstaben versehen.

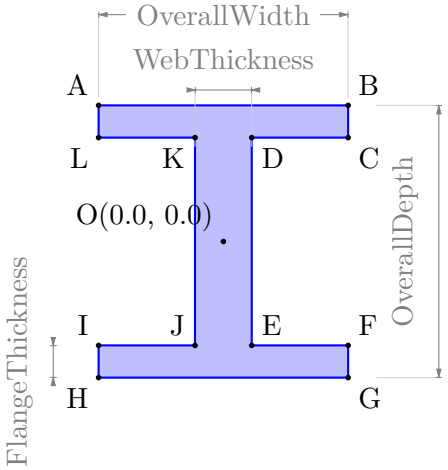


Abbildung 8.15: Darstellung des zu erzeugenden Doppel-T-Profils. Jedem Punkt des Profils wurde ein Buchstabe zugewiesen.

Die Methode createPoint erzeugt für eine gegebene x- und y-Koordinate einen Punkt vom Typ IfcCartesianPoint.

⟨*IShapeProfile createPoint*⟩≡

```
1 private IfcCartesianPoint createPoint(const double x, const double y) {
2    IfcCartesianPoint point = new IfcCartesianPoint();
3
4    point.Coordinates.add(new IfcLengthMeasure(x));
5    point.Coordinates.add(new IfcLengthMeasure(y));
6
7    return point;
8 }
```

Um die Eckpunkte des Doppel-T-Profils zu berechnen, werden zunächst einige Hilfsvariablen eingeführt. Die Variable Left enthält die x-Position des am weitesten links liegenden Punktes. Entsprechend enthalten die Variablen Right, Top und Bottom die x- bzw. y-Positionen der am weitesten rechts, oben und unten liegenden Punkte.

```
⟨IShapeProfile compute points⟩≡
1 double Left = -OverallWidth_ / 2.0;
2 double Right = OverallWidth_ / 2.0;
3 double Top = OverallDepth_ / 2.0;
4 double Bottom = -OverallDepth_ / 2.0;
```

Die Punkte A bis L aus der Abbildung 8.15 werden mit der Methode createPoint erzeugt.

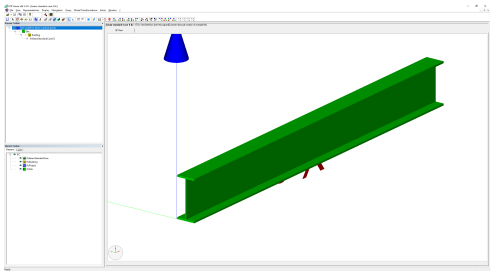
```
⟨IShapeProfile compute points⟩+≡
1  IfcCartesianPoint[] points = new IfcCartesianPoint[12];
2  points[0] = createPoint(Left,Top); // A
3  points[1] = createPoint(Right,Top); // B
4  points[2] = createPoint(Right,Top-FlangeThickness_); // C
5  points[3] = createPoint(WebThickness_*0.5,Top-FlangeThickness_); // D
6  points[4] = createPoint(WebThickness_*0.5,Bottom+FlangeThickness_); // E
7  points[5] = createPoint(Right,Bottom+FlangeThickness_); // F
8  points[6] = createPoint(Right,Bottom); // G
9  points[7] = createPoint(Left,Bottom); // H
10 points[8] = createPoint(Left,Bottom+FlangeThickness_); // I
11 points[9] = createPoint(-WebThickness_*0.5,Bottom+FlangeThickness_); // J
12 points[10] = createPoint(-WebThickness_*0.5,Top-FlangeThickness_); // K
13 points[11] = createPoint(Left,Top-FlangeThickness_); // L
```

Anschließend werden die erzeugten Punkte der Polylinie hinzugefügt:

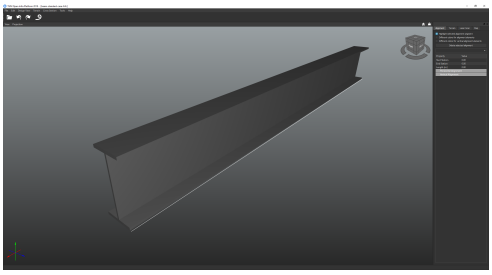
```
⟨IShapeProfile compute points⟩+≡
1 for(int i = 0; i < 12; i++) {
2    polyline.Points.add(points[i]);
3 }
```

Das Profil muss geschlossen werden, deshalb wird der erste Punkt (A) am Ende ein weiteres Mal hinzugefügt.

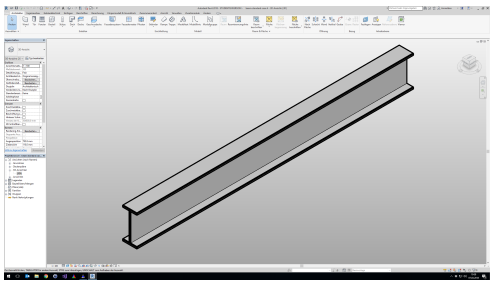
```
⟨IShapeProfile compute points⟩+≡
1 // close profile
2 polyline.Points.add(points[0]); // add point A
```



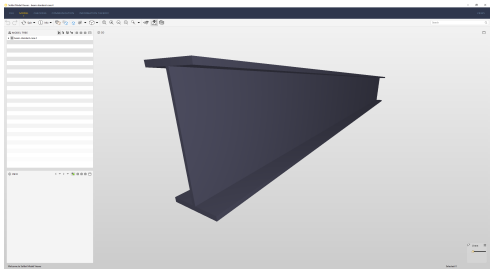
(a) FZK Viewer



(b) TUM Open Infra Platform



(c) Autodesk Revit



(d) Solibri Model Viewer

Abbildung 8.16: Eine parametrische Profilbeschreibung auf Basis der IFC-PL, getestet in verschiedenen Softwareanwendungen

Der vollständige Quelltext der Klasse IShapeProfile befindet sich im Anhang B.3.2. Durch eine Anpassung der IShapeProfile-Klasse könnten auch Ausrundungsparameter unterstützt werden. Dabei würde man die Klasse entsprechend um die Attribute FilletRadius, FlangeEdgeRadius und FlangeSlope erweitern und die Methode getBasicProfileDescription anpassen, so dass diese Ausrundungen beispielsweise mithilfe der Klasse IfcCircle oder IfcBSplineCurve realisiert werden.

8.2.2.4 Integration in Fachapplikationen

Der beschriebene Ansatz erlaubt es, dynamisch neue Profilbeschreibungen in das Datenmodell einzuführen, ohne dass diese explizit standardisiert oder von den Softwareentwicklern umgesetzt werden müssen. Im Rahmen dieser Arbeit wurde dieser Ansatz prototypisch implementiert und mit den Applikationen FZK Viewer, TUM Open Infra Platform, Autodesk Revit sowie Solibri Model Viewer getestet (siehe Abbildung 8.16). Im Testszenario dient das parametrische IFC-PL-Profil als Sweep-Fläche einer IfcExtrudedAreaSolid-Instanz, die genutzt wird, um einen Träger (IfcBeam) geometrisch zu beschreiben.

Abbildung 8.17 zeigt verschiedene Querprofile, die mittels des IFC-PL-Ansatzes erzeugt wurden.

Abbildung 8.18 zeigt den hier vorgestellten Erweiterungsvorschlag der Klassenhierachie der IFC-Profilbeschreibungen mit der Klasse IfcParametricIFCPLProfile-Description. Dieser Ansatz kann vollständig die Klasse IfcParameterizedProfileDef und alle davon abgeleiteten Unterklassen wie z. B. IfcIShapeProfileDef ersetzen.

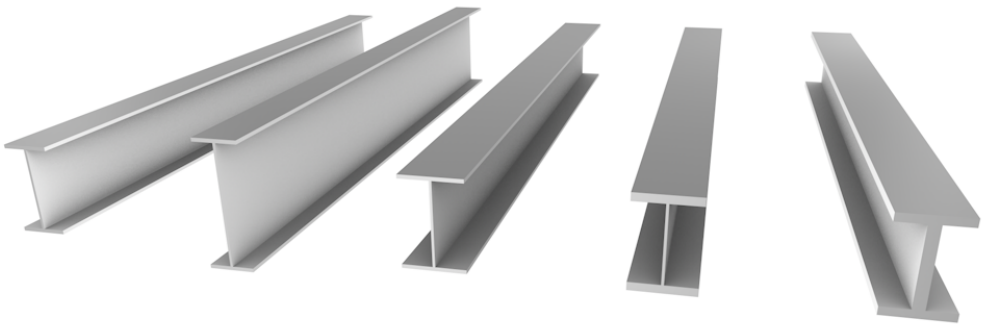


Abbildung 8.17: Verschiedene Querprofile, die mittels des IFC-PL-Ansatzes erzeugt wurden

Damit könnte man die Anzahl der Klassen des IFC-Schemas reduzieren und der zunehmenden Komplexität des Datenmodells entgegenwirken. Natürlich nur unter der Voraussetzung, dass ein Konzept wie die IFC-PL unterstützt wird.

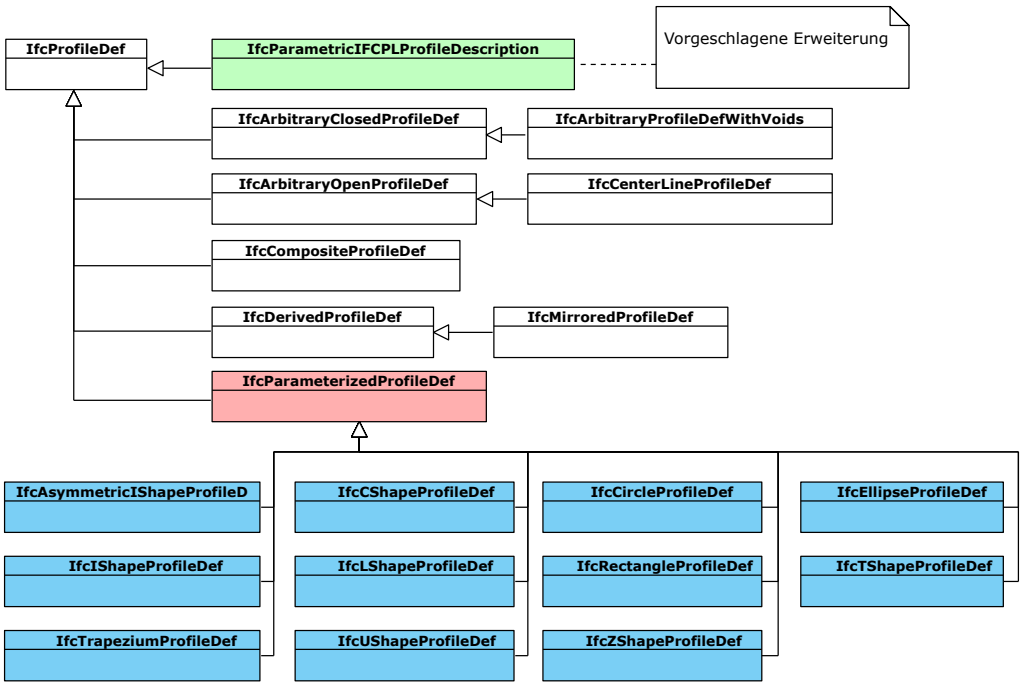


Abbildung 8.18: Erweiterungsvorschlag der Klassenhierarchie der IFC-Profilbeschreibungen mit der Klasse IfcParametricIFCPLProfileDescription als UML-Klassendiagramm dargestellt. Dieser Ansatz kann vollständig die Klasse IfcParameterizedProfileDef und alle davon abgeleiteten Unterklassen (blau dargestellt) wie z. B. IfcIShapeProfileDef ersetzen.

Weiter hilft der Ansatz, wenn eine bestehende Profildefinition verändert oder ergänzt werden soll. Beispielsweise wurde die Klasse IfcIShapeProfileDef von der IFC Version 2x3 auf die Version 4x1 um die Attribute FlangeEdgeRadius und FlangeSlope ergänzt. Diese Attribute existierten in der Version 2.3 nicht. Dies bedeutet für einen Softwarehersteller, der beide IFC-Versionen unterstützten möchte, dass er je nach IFC-Version eine IfcIShapeProfileDef-Instanz unterschiedlich handha-

ben muss. Das IFC-Standardkomitee (Model Support Group) führt sehr ungern Änderungen an bestehenden Entitäten durch, da nach Möglichkeit vermieden werden soll, dass eine Entität in unterschiedlichen Versionen des IFC-Schemas unterschiedlich interpretiert werden muss, weil dies mit einem höheren Aufwand auf Seiten der Softwarehersteller verbunden ist. Zudem wird dadurch die Aufwärtskomptabilität des Datenmodells zerstört, die man ebenfalls gewährleisten will, damit Bestandsdaten möglichst einfach in ein neueres Format überführt werden können. Dies führt im Extremfall dazu, dass man lieber mit Designfehlern im Datenmodell lebt, anstatt diese zu korrigieren. Mit einem IFC-PL-Ansatz könnte man einen Teil dieses Problems umgehen. Der Ansatz erlaubt es, die Daten einer Profilbeschreibung zu ändern, ohne damit die Aufwärtskomptabilität zu zerstören oder Softwareentwickler dazu zu zwingen, verschiedene Versionen einer Profilbeschreibung unterstützen zu müssen.

8.2.3 Rechtwinkliges Kastenwiderlager

In diesem Abschnitt wird die geometrische Beschreibung eines rechtwinkligen Kastenwiderlagers auf Basis der IFC 4.1 mithilfe der IFC-PL beschrieben. Dazu wird eine IFC-PL-Klasse entwickelt, die ein vereinfachtes Widerlager mithilfe einer Menge vordefinierter Parameter beschreibt und abhängig von diesen ein geometrisches Modell auf Basis existierender Elemente des IFC-Datenschemas erzeugt.

Die geometrische Beschreibung des Kastenwiderlagers orientiert sich dabei an den Richtzeichnungen für Ingenieurbauten (RiZ-ING), die Bestandteil der maßgebenden Regelwerke für den Brückenentwurf in Deutschland sind (Geißler, 2014). Die Kreuzungssituation wird der Einfachheit halber als rechtwinklig angenommen, jedoch kann das gezeigte Vorgehen auch auf schiefwinklige Kreuzungssituationen erweitert werden. Das rechtwinklige Kastenwiderlager setzt sich aus der Widerlagerwand, die zur Aufnahme der Auflagerkräfte des Überbaus und der Aufnahme des Erddrucks aus der Hinterfüllung dient, den Flügelwänden, die den seitlich wirkenden Erddruck aus dem Damm aufnehmen, und dem Widerlagerfundament, das die Widerlagerwand und die Flügelwände gründet, zusammen. Für die Anordnung der Flügelwände gibt es verschieden Varianten wie z. B. Parallelflügel, Schrägflügel oder Böschungsflügel. Für das hier beschriebene Beispiel wurden Parallelflügel gewählt. Es gibt ebenfalls verschiedene Gründungsarten. Abhängig vom Baugrundgutachten werden hier typischerweise Flächen- oder Tiefgründungen (z. B. Ortbeton-Bohrpfähle) umgesetzt. Abbildung 8.19 zeigt den Aufbau eines kastenförmigen Widerlagers.

Der Grundriss sowie die Vorder- und Seitenansicht des Kastenwiderlagers sind in Abbildung 8.20 dargestellt.

8.2.3.1 Definition einer Schnittstelle

Ziel dieses Anwendungsfalles ist es, die Geometrie eines vereinfachten Widerlagers zu generieren. Für die geometrische Beschreibung des Widerlagers eignet sich die Klasse IfcSolidModel. Abbildung 8.21 zeigt den Vorschlag einer sehr einfachen

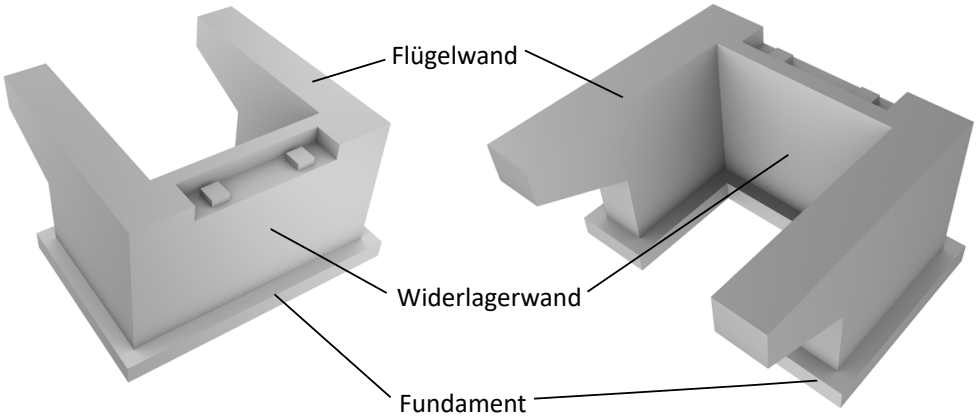
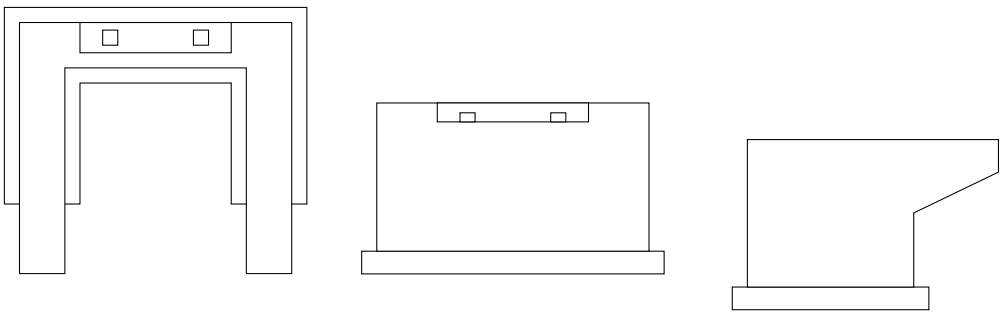


Abbildung 8.19: Exemplarischer Aufbau eines kastenförmigen Widerlagers nach (Geißler, 2014)



(a) Grundriss **(b)** Vorderansicht **(c)** Seitenansicht

Abbildung 8.20: Verschiedene Ansichten des rechtwinkligen Kastenwiderlagers nach (Geißler, 2014)

Schnittstelle (IAbutment) für das Erzeugen einer vereinfachten Widerlagergeometrie. Diese besitzt nur eine Methode namens getSolidModel, deren Zweck es ist, die Geometrie eines Widerlagers als Rückgabewert zu liefern.

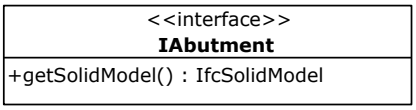


Abbildung 8.21: UML-Klassendiagramm, das die Schnittstelle IAbutment zeigt

Die Schnittstelle wird formal mithilfe der IFC-PL (*IAbutment.ifcpl*) wie folgt beschrieben:

```
1 module IAbutment;
2
3 import IFC4X1.exp;
4
5 interface IAbutment {
6    IfcSolidModel getSolidModel();
7 }
```

8.2.3.2 EXPRESS-Schemaerweiterung

Das IFC 4.1-Schema enthält die Entität IfcCivilElement, die genutzt wird, um Bauelemente wie Widerlager abzubilden. Um den hier beschriebenen Ansatz in das Datenmodell zu integrieren, macht es Sinn, eine neue Entität für Widerlager (IfcAbutment), die auf Basis von IFC-PL definiert werden, in das bestehende Schema einzuführen. Abbildung 8.22 zeigt die Integration der Entität IfcAbutment. Diese erbt direkt von der Klasse IfcCivilElement und besitzt ein Attribut mit dem Namen ConstructionDescription, das vom Typ IfcPLInterfaceRealizationConstructionDescription ist. Dieses speichert den Namen der IFC-PL-Klasse (ClassName) und das Property-Set (InitializationData), mit dem eine entsprechende Instanz der IFC-PL-Klasse initialisiert wird.

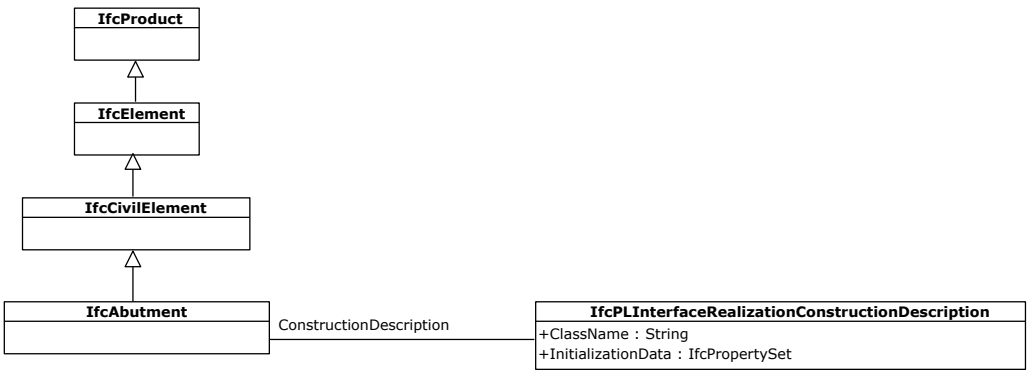


Abbildung 8.22: UML-Klassendiagramm, das die Erweiterung des IFC 4.1-EXPRESS-Schemas zeigt

8.2.3.3 Eine IFC-PL-Klasse für Widerlager

In diesem Abschnitt wird die Implementierung der Schnittelle IAbutment durch die IFC-PL-Klasse Abutment dargestellt. Diese erzeugt eine vereinfachte geometrische Beschreibung der Flügel- und Widerlagerwand. Das vorgestellte IFC-PL-Programm (Abutment) erzeugt dabei eine IFC-Geometrie, die auf Basis der Widerlagerparameter generiert wird. Die Widerlagerparameter werden bei einem Datenaustausch explizit in einer STEP-P21-Datei gespeichert, die generierte IFC-Geometrie aber nicht. Abbildung 8.23 zeigt, auf Basis welcher Parameter die im Folgenden beschriebene Klasse Abutment die Geometrie des Widerlagers definiert. Es gibt Parameter für die Breite der Flügelwände (innerWingWidth), die Breite der Widerlagerwand (innerEndWallWidth), der Gesamtlänge, -breite und -höhe der Widerlagerwand und der Widerlagerflügel (width, length, height) sowie die genauen Abmessungen der Widerlagerflügeldimensionen (wingWidthA, wingWidthB, wingHeightA und wingHeightB).

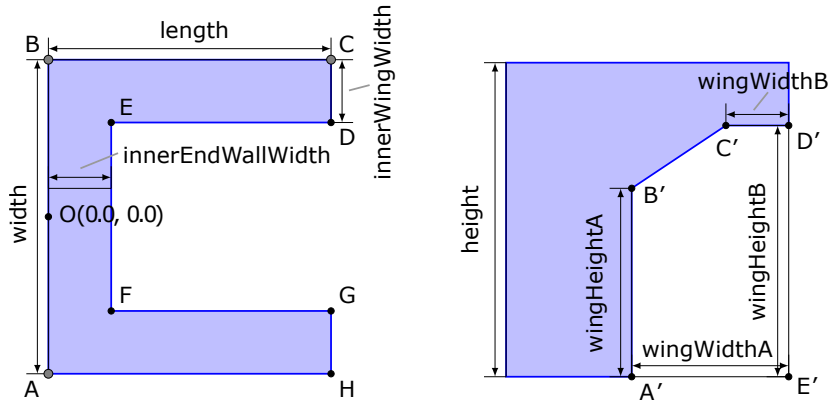


Abbildung 8.23: Drauf- und Seitansicht eines Widerlagers mit entsprechenden Parametern, die die Klasse Abutment verwendet, um die Geometrie eines vereinfachten Widerlagers zu beschreiben

Auf Grundlage dieser Widerlagerparameter wird zunächst ein 2D-Profil der Grundfläche erstellt. Dieses wird senkrecht entsprechend der definierten Höhe (height) nach oben extrudiert. Anschließend wird ein Hilfskörper erstellt, der als Differenzkörper in einer CSG-Operation dient. Der Hilfskörper wird vom vorher erstellten Extrusionskörper abgezogen, um so schließlich die Geometrie der Flügel- und Widerlagerwand zu erhalten. Abbildung 8.24 stellt das beschriebene Vorgehen dar.

Die Abutment-Klasse speichert alle Widerlagerparameter in entsprechenden Member-Variablen. Diese werden mit sinnvollen Default-Werten vorbelegt.

⟨*Abutment class member variables*⟩≡

```
1 private double width_ = 5;
2 private double length_ = 4.5;
3 private double innerWingWidth_ = 1;
4 private double innerEndWallWidth_ = 1;
5 private double height_ = 5;
6 private double wingWidthA_ = 2.5;
7 private double wingWidthB_ = 1.0;
```

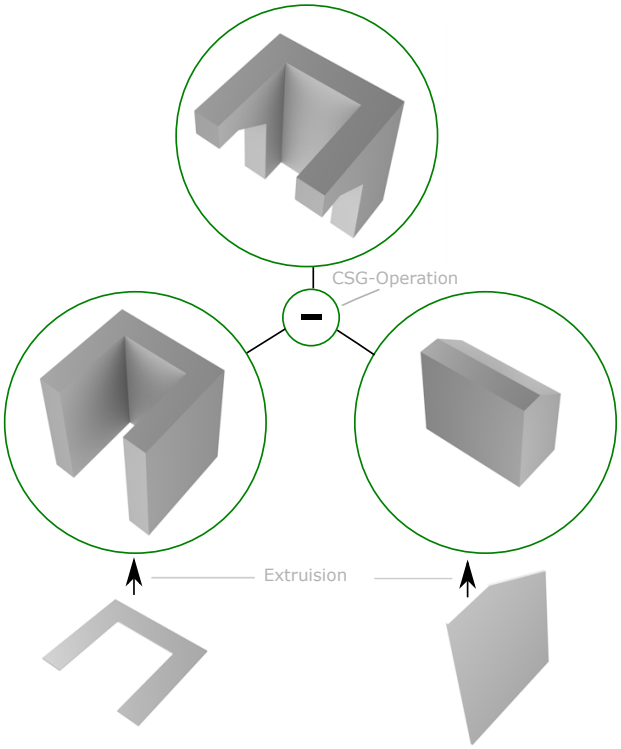


Abbildung 8.24: Übersicht über die Geometrieerzeugung für das Widerlager

```
8  private double wingHeightA_ = 3.0;
9  private double wingHeightB_ = 4.0;
10
11 private double scale_ = 1000;
```

Die globale Einheitenzuordnung (UnitsInContext) eines IFC-Projekts (IfcProject) legt die globalen Einheiten für Kennzahlen und Werte fest, wenn die Einheiten nicht anderweitig über den Entitätstyp definiert sind. Für dieses Beispiel wird angenommen, dass für Längenmessungen die SI-Einheit Meter benutzt wird und entsprechende Maßwerte in Millimeter angegeben werden. Die angegebenen Werte der Widerlagerparameter sind allerdings in der Einheit Meter zu verstehen, da diese intern noch mit dem Skalierungswert (scale_) multipliziert werden. Der Wert von scale_ beträgt 1000 und rechnet damit die Meterangabe in eine Millimeterangabe um[4]. Natürlich kann der Benutzer der Klasse Abutment auch eigene Werte für die Widerlagerparameter mithilfe eines Property-Sets festlegen, das vom Konstruktor der Klasse erwartet wird. Dieser sucht im übergebenen Property-Set nach entsprechenden definierten Eigenschaften und überschreibt gegebenenfalls damit die Default-Werte.

⟨*Abutment class constructor*⟩≡

```
1  public Abutment(IfcPropertySet ps) {
```

[4]Properties können auch selbst definieren, in welcher Einheit diese spezifiziert sind. Dabei wird dann die globale Einheitenzuordnung ignoriert.

```
2     for(int i = 0; i < ps.HasProperties.count(); ++i) {
3        if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
4           IfcPropertySingleValue s =
5              cast<IfcPropertySingleValue>(ps.HasProperties[i]);
6           IfcIdentifier s_id = s.Name;
7
8           readReal(s, "width", width_);
9           readReal(s, "length", length_);
10          readReal(s, "innerWingWidth", innerWingWidth_);
11          readReal(s, "innerEndWallWidth", innerEndWallWidth_);
12          readReal(s, "height", height_);
13          readReal(s, "wingWidthA", wingWidthA_);
14          readReal(s, "wingWidthB", wingWidthB_);
15          readReal(s, "wingHeightA", wingHeightA_);
16          readReal(s, "wingHeightB", wingHeightB_);
17       }
18    }
19 }
```

Die Methode readReal überprüft, ob eine IfcPropertySingleValue-Instanz den gesuchten Eigenschaftsnamen besitzt, und initialisiert gegebenenfalls den übergebenen Parameter mit dem entsprechenden Wert. Sollte beispielsweise die IfcPropertySingleValue-Instanz den Namen width besitzen, so wird der Aufruf readReal(s, "width", width_) dazu führen, dass der Wert (NominalValue) des Properties ausgelesen wird und dem übergebenen Parameter (width_) zugewiesen wird.

⟨*Abutment class methods*⟩≡

```
1  private void readReal(IfcPropertySingleValue ifcPSV, string name,
2                        ref double outValue) {
3     IfcIdentifier s_id = ifcPSV.Name;
4
5     if(s_id.getValue() == name) {
6        IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
7        outValue = ifcReal.getValue();
8     }
9  }
```

Um die Erstellung der Geometrie zu vereinfachen, definiert die Klasse Abutment eine Reihe von Hilfsmethoden. Die Methode createPoint2D ist eine dieser Hilfsmethoden. Diese erzeugt aus einem 2D-Punkt eine IfcCartesianPoint-Instanz.

⟨*Abutment class methods*⟩+≡

```
1  private IfcCartesianPoint createPoint2D(Vector2d v) {
2     IfcCartesianPoint point = new IfcCartesianPoint();
3     point.Coordinates.add(new IfcLengthMeasure(v.x()));
4     point.Coordinates.add(new IfcLengthMeasure(v.y()));
5     return point;
6  }
```

Eine weitere Hilfsmethode ist createPoint3D, die anlog zum 2D-Fall für einen gegebenen 3D-Punkt eine IfcCartesianPoint-Instanz erzeugt.

⟨*Abutment class methods*⟩+≡

```
1  private IfcCartesianPoint createPoint3D(Vector3d v) {
2      IfcCartesianPoint point = new IfcCartesianPoint();
3      point.Coordinates.add(new IfcLengthMeasure(v.x()));
4      point.Coordinates.add(new IfcLengthMeasure(v.y()));
5      point.Coordinates.add(new IfcLengthMeasure(v.z()));
6      return point;
7  }
```

Eine ähnliche Funktionalität wird für das Erstellen einer IfcDirection-Entität benötigt, die aus einem 3D-Vektor eine solche Entität erzeugt.

```
⟨Abutment class methods⟩+≡
1  private IfcDirection createDirecton(double x, double y, double z) {
2      IfcDirection dir = new IfcDirection();
3      dir.DirectionRatios.add(new IfcReal(x));
4      dir.DirectionRatios.add(new IfcReal(y));
5      dir.DirectionRatios.add(new IfcReal(z));
6      return dir;
7  }
```

Die Grundfläche des Widerlagers wird auf Basis einer IfcArbitraryClosedProfileDef-Entität repräsentiert. Die Entität IfcArbitraryClosedProfileDef besitzt ein Attribut mit dem Namen OuterCurve, das auf Basis der Klasse IfcCurve erlaubt, den Rand der IfcArbitraryClosedProfileDef-Instanz zu beschreiben. Um den Rand der Grundfläche zu spezifizieren, wird eine IfcPolyline-Instanz verwendet. IfcPolyline ist eine Unterklasse von IfcCurve und kann daher dem Attribut OuterCurve zugewiesen werden.

```
⟨Abutment class methods⟩+≡
1  private IfcArbitraryClosedProfileDef createGroundArea() {
2      IfcPolyline polyline = new IfcPolyline();
3
4      ⟨Abutment class compute ground area points 171⟩
5
6      IfcArbitraryClosedProfileDef closedProfile =
7          new IfcArbitraryClosedProfileDef();
8      closedProfile.OuterCurve = polyline;
9      closedProfile.ProfileType =
10         new IfcProfileTypeEnum(IfcProfileTypeEnum.AREA);
11
12     return closedProfile;
13 }
```

Die Punkte der Grundfläche wurden in Abbildung 8.23 mit den Buchstaben A bis H bezeichnet. Diese werden nun entsprechend den gegebenen Widerlagerparametern berechnet und anschließend in die Polylinie eingefügt. Dabei werden die Punkte auch entsprechend mit dem vorher genannten Skalierungsfaktor scale_ multipliziert.

```
⟨Abutment class compute ground area points⟩≡
1  Vector2d[] points = new Vector2d[9];
2
```

```
3  /*A*/ points[0] = new Vector2d(0, -width_ / 2.0);
4  /*B*/ points[1] = new Vector2d(0, width_ / 2.0);
5  /*C*/ points[2] = new Vector2d(length_, width_ / 2.0);
6  /*D*/ points[3] = new Vector2d(length_, width_ / 2.0 - innerWingWidth_);
7  /*E*/ points[4] = new Vector2d(innerEndWallWidth_,
8                       width_ / 2.0 - innerWingWidth_);
9  /*F*/ points[5] = new Vector2d(innerEndWallWidth_,
10                      -width_ / 2.0 + innerWingWidth_);
11 /*G*/ points[6] = new Vector2d(length_, -width_ / 2.0 + innerWingWidth_);
12 /*H*/ points[7] = new Vector2d(length_, -width_ / 2.0);
13
14 // The OuterCurve has to be a closed curve.
15 points[8] = new Vector2d(0, -width_ / 2.0); // A
16
17 for(int i = 0; i < 9; i++) {
18    polyline.Points.add(createPoint2D(scale_ * points[i]));
19 }
```

Die Grundfläche wird in der Methode createBaseMesh senkrecht nach oben extrudiert, wie dies in Abbildung 8.24 im linken Teilbaum der CSG-Operation zu sehen ist. Dabei wird die Entität IfcExtrudedAreaSolid verwendet, die als Sweep-Fläche die Grundfläche des Widerlagers verwendet. Die Extrusionsrichtung (ExtrudedDirection) sowie die Tiefe der Extrusion (Depth) werden entsprechend festgelegt.

⟨*Abutment class methods*⟩+≡

```
1  private IfcExtrudedAreaSolid createBaseMesh() {
2     IfcExtrudedAreaSolid baseMesh = new IfcExtrudedAreaSolid();
3     baseMesh.SweptArea = createGroundArea();
4     baseMesh.ExtrudedDirection = createDirecton(0, 0, 1);
5     baseMesh.Depth = new IfcPositiveLengthMeasure(scale_ * height_);
6     return baseMesh;
7  }
```

Abbildung 8.24 zeigt im rechten Teilbaum der CSG-Operation einen Hilfskörper, der später von der Hauptgeometrie abgezogen werden soll. Dieser Hilfskörper wird erzeugt, indem zunächst seine Fläche durch die Methode createAuxArea erstellt wird. Die Fläche der Hilfsgeometrie wird ebenfalls wieder mittels einer IfcArbitraryClosedProfileDef-Instanz beschrieben. Der Rand der Fläche wird ebenfalls durch eine Polylinie definiert.

⟨*Abutment class methods*⟩+≡

```
1  private IfcArbitraryClosedProfileDef createAuxArea() {
2     IfcPolyline polyline = new IfcPolyline();
3
4     ⟨Abutment class compute aux mesh area points⟩
5
6     IfcArbitraryClosedProfileDef closedProfile = new
        IfcArbitraryClosedProfileDef();
7     closedProfile.OuterCurve = polyline;
8     closedProfile.ProfileType = new
        IfcProfileTypeEnum(IfcProfileTypeEnum.AREA);
```

```
9
10      return closedProfile;
11  }
```

Die Punkte der Fläche des Hilfskörpers sind in Abbildung 8.23 mit den Buchstaben A' bis E' bezeichnet. Diese werden entsprechend den gegebenen Widerlagerparametern berechnet und anschließend in die Polylinie eingefügt. Dabei werden die Punkte ebenfalls mit dem Skalierungsfaktor scale_ multipliziert.

```
⟨Abutment class methods⟩≡
1  Vector2d[] points = new Vector2d[6];
2
3  /* A' */ points[0] = new Vector2d(length_ - wingWidthA_, 0);
4  /* B' */ points[1] = new Vector2d(length_ - wingWidthA_, wingHeightA_);
5  /* C' */ points[2] = new Vector2d(length_ - wingWidthB_, wingHeightB_);
6  /* D' */ points[3] = new Vector2d(length_, wingHeightB_);
7  /* E' */ points[4] = new Vector2d(length_, 0.0);
8  // The OuterCurve has to be a closed curve.
9  /* A' */ points[5] = new Vector2d(length_ - wingWidthA_, 0);
10
11 for(int i = 0; i < 6; i++) {
12     polyline.Points.add(createPoint2D(scale_ * points[i]));
13 }
```

Der Hilfskörper entsteht durch Extrusion der durch die Methode createAuxArea erstellten Fläche. Dabei wird die Fläche entlang der Richtung der negativen y-Achse extrudiert. Um dies zu erreichen, wird in der Methode createAuxiliaryMesh eine entsprechende IfcAxis2Placement3D-Instanz erzeugt, die die Extrusionsrichtung entsprechend festlegt.

```
⟨Abutment class methods⟩+≡
1  private IfcExtrudedAreaSolid createAuxiliaryMesh() {
2      IfcAxis2Placement3D placement = new IfcAxis2Placement3D();
3      placement.Location = createPoint3D(new Vector3d(0, scale_ * width_ /
         2.0, 0));
4      placement.Axis = createDirecton(0.0, -1.0, 0.0);
5
6      IfcExtrudedAreaSolid baseMesh = new IfcExtrudedAreaSolid();
7      baseMesh.SweptArea = createAuxArea();
8      baseMesh.ExtrudedDirection = createDirecton(0, 0, 1);
9      baseMesh.Depth = new IfcPositiveLengthMeasure(scale_ * height_);
10     baseMesh.Position = placement;
11     return baseMesh;
12 }
```

Im letzten Schritt wird die Methode getSolidModel definiert, die durch das Interface IAbutment vorgegeben ist. Dabei wird eine boolesche Differenzoperation definiert, wobei als erster Operand die extrudierte Grundfläche des Widerlagers gewählt wird, die durch die Methode createBaseMesh zurückgeliefert wird, und als zweiter Operand der extrudierte Hilfskörper gewählt wird, der durch die Methode createAuxiliaryMesh berechnet wird. Die Entität IfcCsgSolid ist eine Unter-

klasse von IfcSolidModel und erhält als Attributwert die vorher definierte CSG-Operation. Abbildung 8.23 veranschaulicht die Differenzoperation und zeigt die so entstehende Geometrie im Wurzelknoten.

⟨*Abutment class methods*⟩+≡

```
1  public IfcSolidModel getSolidModel() {
2      IfcBooleanResult br = new IfcBooleanResult();
3      br.Operator = new IfcBooleanOperator(IfcBooleanOperator.DIFFERENCE);
4      br.FirstOperand = createBaseMesh();
5      br.SecondOperand = createAuxiliaryMesh();
6
7      IfcCsgSolid csgSolid = new IfcCsgSolid();
8      csgSolid.TreeRootExpression = br;
9
10     return csgSolid;
11 }
```

Die IFC-PL-Klasse Abutment erbt vom Interface IAbutment und setzt sich aus dem Klassenkonstruktor, Membermethoden und Membervariablen zusammen. Die Klasse selbst wird innerhalb des Moduls Abutment definiert und innerhalb der Datei Abutment.ifcpl mit den nötigen import-Anweisungen versehen.

⟨*Abutment.ifcpl*⟩≡

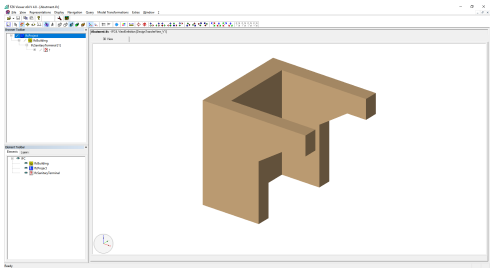
```
1  module Abutment;
2
3  import Core;
4  import Math;
5  import IAbutment;
6  import IFC4X1.exp;
7
8  class Abutment : IAbutment {
9      ⟨Abutment class constructor 169⟩
10     ⟨Abutment class methods 170⟩
11     ⟨Abutment class member variables 168⟩
12 }
```

Der vollständige Quelltext der Klasse Abutment befindet sich im Anhang B.3.3.

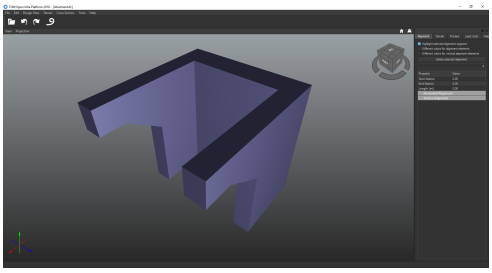
8.2.3.4 Integration in Fachapplikationen

Abbildung 8.25 zeigt die Integration der IAbutment-Schnittstelle in verschiedene Softwareanwendungen. Hierbei ist es nützlich, dass die bestehenden Softwareprodukte jeweils mit den verwendeten IFC-Entitäten zur Beschreibung von Geometrie umgehen können, wie z. B. mit IfcCsgSolid oder IfcExtrudedAreaSolid, und das IFC-Datenmodell sozusagen den gemeinsamen herstellerneutralen Nenner dieser Applikationen darstellt. Im Rahmen eines Datenaustausches werden dabei lediglich Widerlager-Attribute mittels einer STEP-P21-Datei übergeben, aber nicht die CSG-Geometrie selbst. Letztere wird intern durch das empfangende Programm (mittels IFC-PL-Programm und IFC-PL-Laufzeitumgebung) generiert.

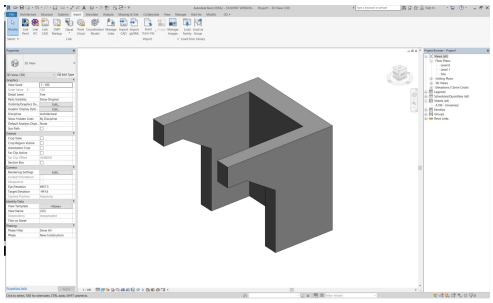
Die vorgestellte Schnittstelle bildet nur die geometrische Seite eines Widerlagers ab. Die entwickelte IFC-PL-Klasse Abutment erzeugt dabei nur die Geometrie



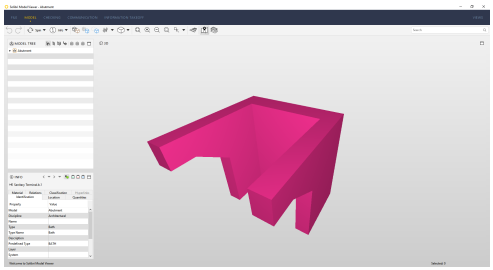
(a) FZK Viewer



(b) TUM Open Infra Platform



(c) Autodesk Revit



(d) Solibri Model Viewer

Abbildung 8.25: Integration der IAbutment-Schnittstelle in verschiedene Softwareanwendungen

für die Flügel- und die Widerlagerwand. Das Fundament oder Detailmodellierungen des Auflagers werden dabei ignoriert. Zudem beschränkt sich die hier entwickelte Klasse auf ein rechtwinkliges Kreuzungsszenario. Dies sind zwar Limitierungen, welche die hier gezeigte Abutment-Klasse besitzt, jedoch handelt es sich hierbei nicht um prinzipielle Limitierungen des IFC-PL-basierten Ansatzes. Der IFC-PL-Programmcode könnte noch erweitert werden, so dass dieser auch Geometrieelemente für das Fundament erzeugt oder analog die Detailgeometrie der Auflager generiert. Genauso könnte man noch nutzerfreundliche Parameter einführen, z. B. einen Böschungs- und Kreuzungswinkel, und so auch schiefwinklige Kreuzungssituationen unterstützen. Weiterhin sieht RiZ-ING für die Flügelausbildung zwei Varianten vor, die in der RiZ-ING als Regelfall bzw. als Variante bezeichnet werden. Hier könnte man durch eine Enumeration und eine if-Abfrage beide Umsetzungen der Flügelausbildung unterstützen. Weiter ist es denkbar, dass man beispielsweise auf Basis einer überführten Trasse, die auf Basis der Entität IfcAlignment definiert ist, den Kreuzungswinkel automatisch berechnet. Die Möglichkeiten, die hier der IFC-PL-Ansatz bietet, sind sehr umfangreich.

Abbildung 8.26 zeigt, wie der beschriebene Ansatz auf Brückenfundamente und auf den Überbau von Brücken ausgeweitet werden kann. Dabei können auch Regeln, wie z. B. die Anzahl der Brückenpfeiler in Abhängigkeit der Brückenlänge zu wählen ist, berücksichtigt werden, wie ebenfalls in Abbildung 8.26 und im folgenden Ausschnitt eines IFC-PL-Programms dargestellt ist.

```
1 [...]
2 void createPier() {
```

```
3     if (length_ > 60){
4         double pierThickness = 5.0;
5         double offset = 0.5;
6
7         int numberOfFields = length / 30;
8         for (int i = 0; i < numberOfFields-1; ++i) {
9             Pier pier = new Pier();
10            [...]
11        }
12    [...]
13 }
14 [...]
```

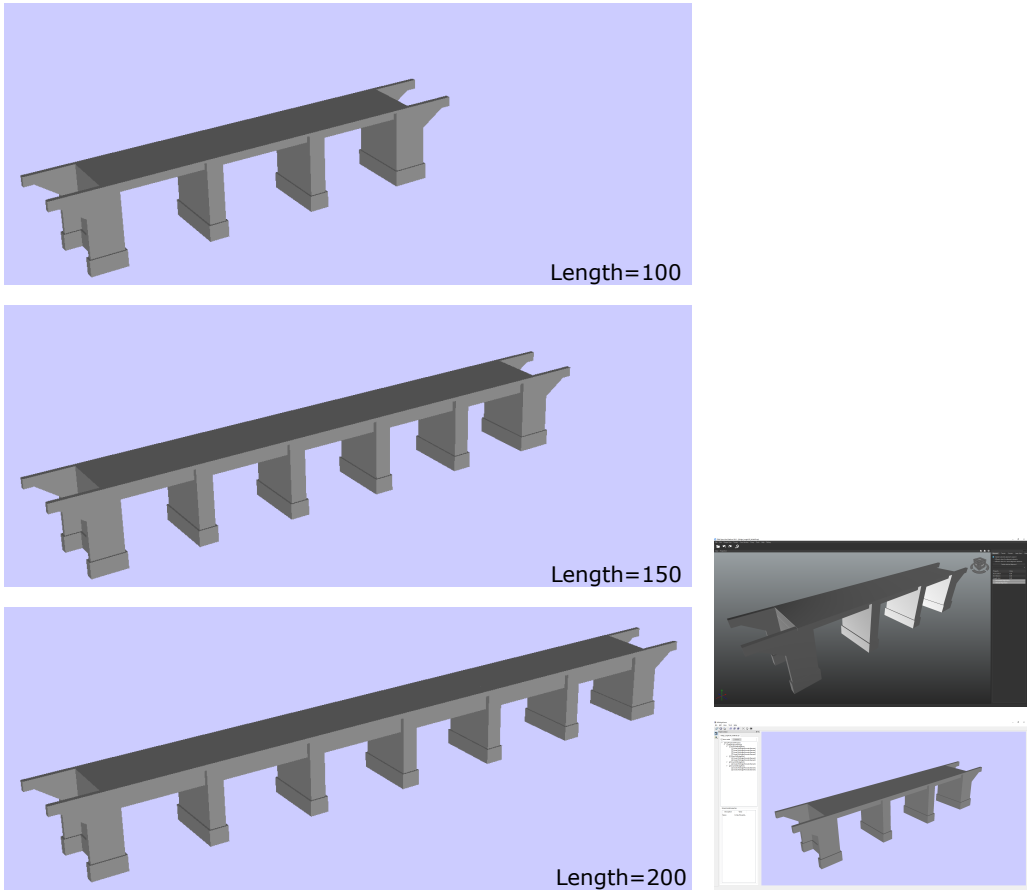


Abbildung 8.26: Erweiterung des beschriebenen IFC-PL-Ansatzes auf Brückenfundamente und auf den Überbau von Brücken. Brückenpfeiler werden in Abhängigkeit der Brückenlänge generiert.

Das Interface IAbutment kann natürlich auch noch stark verbessert werden, damit eine Zielapplikation bessere Möglichkeiten hat, die erzeugten Daten im Nachhinein zu modifizieren. Beispielsweise könnte ein erweitertes Interface Methoden anbieten, mit denen man die Widerlagerwand explizit als IfcWall erhält. Ähnlich könnte man mit den anderen Bauteilen des Widerlagers vorgehen. Zudem könnte

ein solcher Ansatz auch das Level of Development, das sich in Level of Geometry und Level of Information aufteilt (Hausknecht & Liebich, 2016), berücksichtigen.

```
1  enum eLevelOfGeometry {
2      LoG_100,
3      LoG_200,
4      LoG_300,
5      LoG_400,
6      LoG_500
7  };
8
9  // Extended IAbutment interface
10 interface IAbutment {
11     []
12     IfcWall getAbutmentWall(eLevelOfGeometry lod);
13     IfcWall getWingWall(eLevelOfGeometry lod, int index);
14     IfcFooting getFoundation(eLevelOfGeometry lod);
15     IfcSolidModel getBearing(eLevelOfGeometry lod);
16     []
17 }
```

Der hier vorgestellte Ansatz eignet sich in ähnlicher Weise für die Unterstützung von herstellerspezifischen BIM-Objekten wie beispielsweise Türen oder Fenster. Hier könnten analog auf Basis beliebiger Parameter, die in einem IFC-Propery-Set verwaltet werden, beliebige Geometrieelemente, wie in Abbildung 8.27 dargestellt, erzeugt und dadurch intelligente BIM-Objekte realisiert werden.

Abbildung 8.27: Auf Basis des gezeigten Ansatzes könnte beispielsweise auch ein herstellerspezifischer Fenstergenerator implementiert werden, der auf Basis von Eigenschaftswerten unterschiedliche Geometrien erzeugt.

Für den Widerlager-Import in Programmen, die keine Unterstützung für die Integration eines IFC-PL-Ansatzes bieten, jedoch IFC 4.1 unterstützen, gibt es folgende Lösung: Mittels eines IFC-PL-kompatiblen Programms, das IFC 4.1-Exportmöglichkeiten bietet, können die mittels IFC-PL generierten Geometriedaten als IFC 4.1-Daten exportiert werden. Diese Daten können dann über den IFC 4.1-Import der Zielanwendung importiert werden. Da die generierten Geometriedaten auf Basis des Abutment-Schemas kompatibel zu IFC 4.1 sind (es wurden keine Änderungen am Schema bezüglich der Geometriedaten vorgenommen), ist es ebenfalls möglich, auf Basis des Standalone-Modus von IFC-PL eine Exportfunktionalität zu realisieren, die die Geometriedaten als IFC 4.1-Datei ausgibt.

8.3 Prüfung von Normen und Richtlinien

Im Bereich des Bauwesens sind eine Vielzahl von Normen und Richtlinien anzutreffen, die der Vereinheitlichung von Anordnungen im Bauwesen und der Sicherung von etablierten Technikstandards dienen (Preidel *et al.*, 2015). Die manuelle Überprüfung dieser Richtlinien und Normen auf Konformität ist fehleranfällig und arbeitsintensiv. In den letzten vier Jahrzenten wurden deshalb im Bereich der automatisierten und halbautomatischen Konformitätsprüfung für das Bauwesen umfangreiche Forschungsarbeiten durchgeführt (Dimyadi & Amor, 2013). Dabei wurden regelbasierte Systeme entwickelt, die auf Grundlage eines Gebäudeinformationsmodells wie beispielsweise der IFC Richtlinien und Standards automatisiert oder zumindest teilautomatisiert überpüfen können. In (Eastman *et al.*, 2009) wird der Prozess, der eine automatisierte Überprüfung von Regeln, basierend auf Normen, Regelwerken, Standards, Richtlinien oder ähnlichen Werken, ermöglicht, in vier Teilbereiche unterteilt. Abbildung 8.28 zeigt diese Aufteilung. Im ersten Schritt erfolgt die Regelinterpretation. Dabei geht es darum, dass Wissen, das informal in Texten und Abbildungen in verschiedenen Regelwerken beschrieben ist, zu formalisieren, so dass es maschineninterpretierbar wird. Ziel des Schritts ist es, Regeln zu gewinnen, die automatisiert abgeprüft werden können. Damit bestimmte Regeln anwendbar sind, muss im zweiten Schritt das Bauwerksinformationsmodell erst entsprechend aufbereitet werden. Um beispielsweise zu prüfen, ob die Statik eines Bauwerks die geforderten Anforderungen erfüllt, kann in einem Vorverarbeitungsschritt auf Basis der Daten des Bauwerksinformationsmodells erst eine FEM-Simulation durchgeführt werden, deren Ergebnisse in das aufbereitete Modell einfließen. Darauf basierend kann nun im dritten Schritt die Regelprüfung erfolgen. Dabei werden die automatisiert überprüfbaren Regeln gegen das aufbereitete Modell geprüft und die Ergebnisse dieser Prüfung festgehalten. Im letzten Schritt erfolgt die Ergebnisauswertung. Dabei werden die ermittelten Ergebnisse an den Benutzer eines solchen Systems zurückgemeldet. Hier wäre z. B. eine textuelle Auflistung bzw. Beschreibung der verletzten Regeln oder eine grafische 3D-Darstellung der Bauteilelemente, die entsprechende Regeln verletzen, denkbar.

Eine gewisse Vorreiterstellung im Bereich der automatisierten Regelkonformitätsüberprüfung (Automated Code Compliance Checking) nimmt der Stadtstaat Singapur ein (Preidel & Borrmann, 2015). Dieser führte im Jahr 1995 die Plattform CORENET (Construction and Real Estate Network) ein. CORENET dient dabei als Kollaborationsplattform zwischen den Baubeteiligten, insbesondere den beteiligten singapurischen Behörden (Lin, 1995). CORENET setzt sich aus verschiedenen Modulen zusammen. Für den Bereich Automated Code Compliance Checking spielt hier das Modul e-PlanCheck eine wichtige Rolle. e-PlanCheck bietet automatisierte Prüf-Funktionalitäten, die sich auf die für Singapur national gültigen Vorschriften konzentrieren, beispielsweise Regulatorien bezüglich barrierefreier Zugänge oder etwa Brandschutzverordnungen (Xu *et al.*, 2004). Seit dem Jahr 1998 arbeitet das e-PlanCheck-Modul auf Basis von Gebäudeinformationsmodellen, die mit den IFC beschrieben sind (Eastman *et al.*, 2009). Laut den Autoren des Papers (Eastman *et al.*, 2009) nutzen mehr als 2500 Firmen

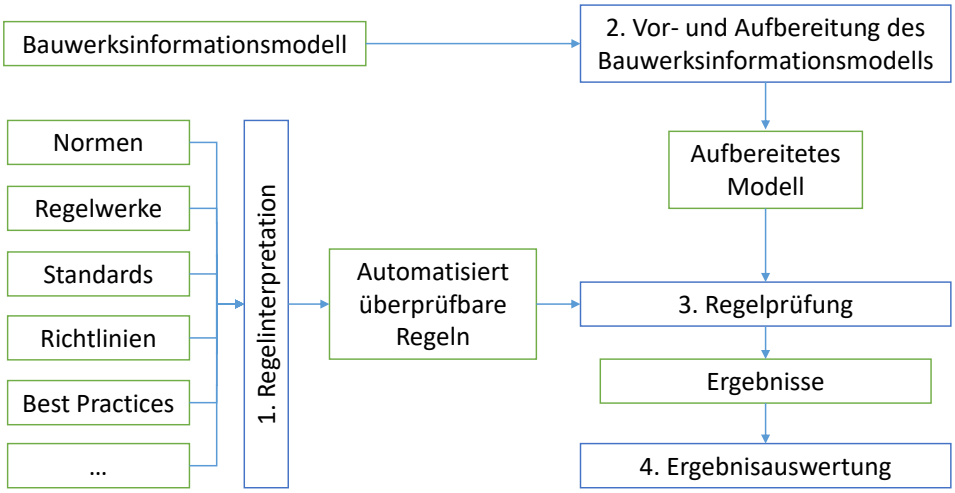


Abbildung 8.28: Struktur und Bestandteile eines regelbasierten Systems zur Modellüberprüfung, angelehnt an (Preidel *et al.*, 2015) und (Eastman *et al.*, 2009)

die CORENET-Plattform für die Einreichung von Plänen bei den singapurischen Behörden. Die große Anwenderzahl erklärt sich daraus, dass Baugenehmigungen in Singapur an die Verwendung der CORENET-Plattform gekoppelt sind. Nur Baupläne, die von e-PlanCheck akzeptiert werden, können eine Baugenehmigung erhalten (Preidel & Borrmann, 2015). Durch die starke Automatisierung der Planüberprüfung können Baugenehmigungen jedoch zügig bearbeitet werden. Allerdings hat Singapur aufgrund seiner speziellen politischen, wirtschaftlichen und demographischen Eigenschaften gute Voraussetzungen, um ein solches System durchzusetzen. Die Übertragbarkeit des singapurischen Modells auf andere Länder ist daher fraglich (Preidel & Borrmann, 2015).

8.3.1 Black-Box- und White-Box-Ansätze

Im Bereich der regelbasierten Systeme zur Modellüberprüfung unterscheidet man zwischen Black-Box- und White-Box-Ansätzen (Preidel & Borrmann, 2015). In einem Black-Box-Ansatz ist die interne Funktionsweise des regelbasierten Systems zur Modellüberprüfung vom Benutzer verborgen. Der Benutzer stellt hier als Eingabe nur ein Bauwerksinformationsmodell bereit und erhält als Ausgabe die Ergebnisauswertung des Systems. Im Gegensatz dazu werden bei einem White-Box-Ansatz auch die Interna der Modellprüfung und die einzelnen Schritte, die für eine Regelprüfung notwendig sind, für den Benutzer offengelegt und einsehbar gemacht. Beim CORENET e-PlanCheck-Modul handelt es sich um einen Black-Box Ansatz. In (Preidel & Borrmann, 2015) wird ein White-Box-Ansatz, basierend auf einer visuellen Programmiersprache, vorgestellt. Beide Ansätze bieten Vor- und Nachteile. Abgeschlossene Systeme (Black-Box-Ansatz) haben beispielsweise den Vorteil, dass diese nur gering fehleranfällig gegenüber Benutzereingaben sind, da dieser im Wesentlichen nur ein Bauwerksinformationsmodell als Eingabe bereit-

stellen muss. Gleichzeitig fehlt dadurch jedoch dem Benutzer die Nachvollziehbarkeit der Regelprüfung, da die einzelnen Schritte der Regelprüfung nicht eingesehen werden können. Im Gegensatz dazu können bei einem White-Box-Ansatz die Interna der Modellprüfung eingesehen werden. Eine komplexe Regelprüfung kann jedoch sehr schnell unübersichtlich werden. Für weitere Vor- und Nachteile beider Ansätze sei hier auf (Preidel *et al.*, 2015) verwiesen.

White-Box-Ansätze verwenden im Regelfall eine Code Representation Language, also eine Sprache, die verwendet wird, um Regeln formal zu definieren, so dass diese gegen ein Modell geprüft werden können. Als Beispiel einer Code Representation Language kann hier die Building Environment and Analysis Language (BERA) angeführt werden (Lee *et al.*, 2014). Diese Sprache kann natürlich auch visueller Natur sein, wie z. B. in (Preidel & Borrmann, 2016) beschrieben. In (Yurchyshyna & Zarli, 2009) wird als Code Representation Language die Abfragesprache SPARQL verwendet.

8.3.2 Richtlinien für die Anlage von Landstraßen

Im Folgenden soll gezeigt werden, wie die Sprache IFC-PL als Code Representation Language und die IFC-PL-Umgebung als regelbasiertes System zur Modellüberprüfung eingesetzt werden können. Dies wird exemplarisch anhand einer Regel aus den Richtlinien für die Anlage von Landstraßen (FGSV, 2012) durchgeführt.

Die RAL ist Teil der Regelwerke des Straßenentwurfs (R1-Regelwerke) und dient als wesentliche Planungsgrundlage für Landstraßen. Die Mehrzahl der Richtlinien sind keine Muss-, sondern überwiegend Soll-Vorgaben. Die Richtlinien für die Anlage von Landstraßen (RAL) sollen den Entwurf von sicheren und funktionsgerechten Landstraßen gewährleisten und standardisieren (FGSV, 2012). Diese basieren auf Erkenntnissen mehrerer Projekte aus dem Forschungsprogramm des Bundesministeriums für Verkehr, Bau und Stadtentwicklung (BMVBS) und der Bundesanstalt für Straßenwesen (BASt). Sie stellt die Grundlage für alle Planungen und Entwürfe für den Neubau sowie den Um- und Ausbau von Landstraßen, die im Verantwortungsbereich des Bundes liegen, dar. Die RAL unterscheidet vier verschiedene Entwurfsklassen. Die Entwurfsklasse legt Vorgaben für unterschiedliche Bereiche fest, wie etwa den Regelquerschnitt der Fahrbahn, die Ausstattung der Fahrbahn (Markierungen, vertikale Verkehrszeichen, wegweisende Beschilderung usw.) oder die Führung von Fußgängern und Radfahrern. Außerdem enthält die RAL Richtlinien für die räumliche Linienführung. Tabelle 8.1 zeigt empfohlene Radien und Mindestlängen von Kreisbögen der RAL für verschiedene Entwurfsklassen.

Für das hier beschriebene Beispiel wird angenommen, dass die Trassierungdaten als IFC 4.1-Datei vorliegen und es sich bei der Trassierung um die Entwurfsklasse 1 (EKK 1) handelt. Abbildung 8.29 zeigt die für dieses Beispiel relevanten Klassen aus dem IFC 4.1-Schema.

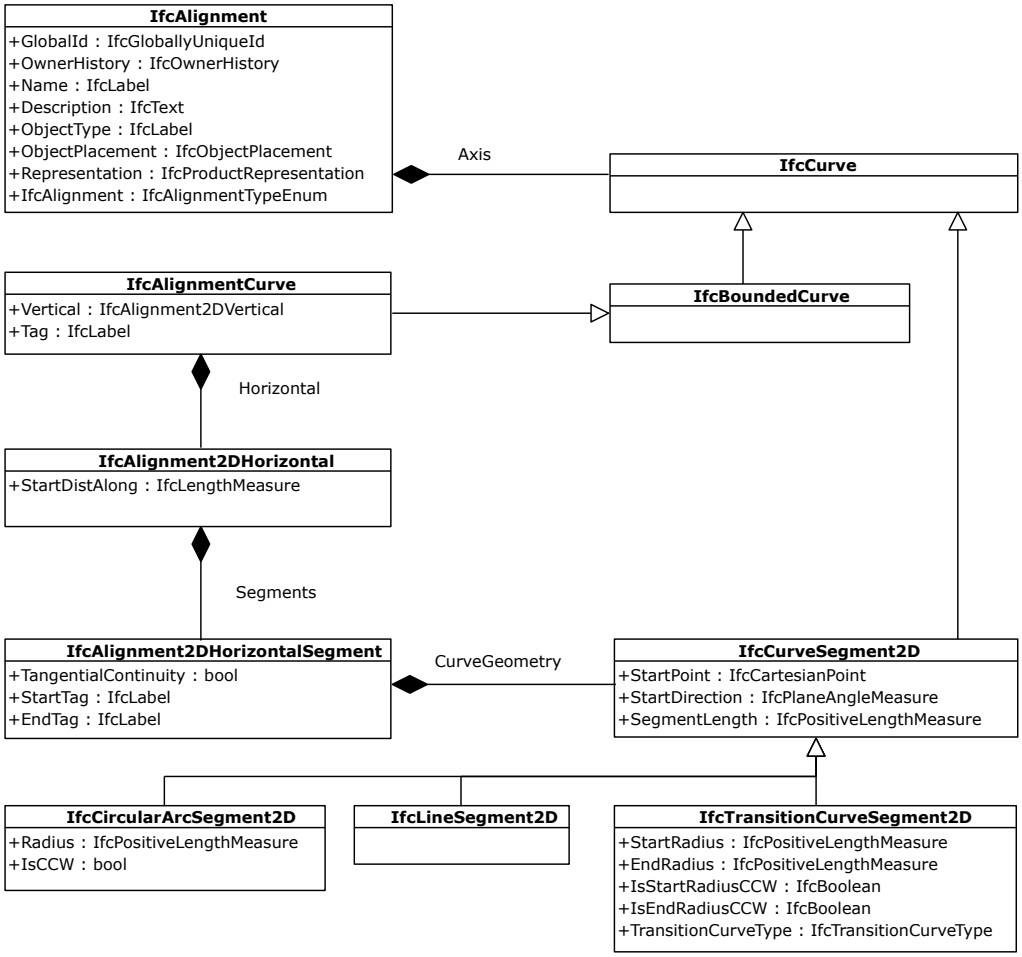


Abbildung 8.29: Relevate Klassen für die Trassierung aus dem IFC 4.1-Schema, dargestellt als UML-Klassendiagramm

Tabelle 8.1: Empfohlene Radien und Mindestlängen von Kreisbögen der RAL

Entwurfsklasse	**Radienbereiche R [m]**	**Mindestlänge L [m]**
EKL 1	≥ 500	70
EKL 2	400 - 900	60
EKL 3	300 - 600	50
EKL 4	175 - 300	40

8.3.2.1 Das Hauptprogramm

Im Folgenden wird schrittweise ein IFC-PL-Programm entwickelt, das diese Prüfung auf Basis einer Eingabedatei durchführt und die Ergebnisse an den Benutzer zurückmeldet. Dieses Programm wird im Folgenden als RALChecker bezeichnet. Das Programm soll sich im gleichnamigen Modul befinden.

⟨*RALChecker.ifcpl*⟩≡
```
1 module RALChecker;
```

Für Ausgaben wird das Core-Modul eingebunden. Da unsere Eingabedatei im Format IFC 4.1 vorliegt, wird auch das entsprechende Schema importiert.

⟨*RALChecker.ifcpl*⟩+≡
```
1 import Core;
2 import IFC4X1.exp;
```

Das Programm soll im Standalone-Modus (siehe Abschnitt 5.3) betrieben werden, daher wird eine main-Funktion definiert. Es wird angenommen, dass der Benutzer als Eingabe den Dateipfad zu einer IFC 4.1-STEP-Datei angibt. Daher wird die main-Funktion verwendet, die als Parameter Kommandozeilenargumente in Form eines string-Arrays erwartet.

⟨*RALChecker.ifcpl*⟩+≡
```
1 void main(string[] args) {
2     ⟨RALChecker main function body 182⟩
3 }
```

Im nächsten Schritt wird geprüft, ob der Benutzer genau ein Kommandozeilenargument bereitgestellt hat. Falls nicht, wird eine Meldung an den Benutzer ausgegeben und der Programmablauf beendet.

⟨*RALChecker main function body*⟩≡
```
1 if(args.count() != 1) {
2     print("Program called with invalid number of arguments");
3     return;
4 }
```

Der Dateiname, der als Kommandozeilenargument übergeben wurde, wird der Variablen filename zugewiesen.

⟨*RALChecker main function body*⟩+≡
```
1 string filename = args[0];
```

Es wird geprüft, ob die Datei existiert und gegebenenfalls eine Warnung an den Benutzer ausgegeben und der Programmablauf beendet.

⟨*RALChecker main function body*⟩+≡
```
1  if(!exists(filename)) {
2      print("File does not exist.");
3      return;
4  }
```

Mithilfe der implizit vorhandenen Klassen IfcStepReader und Model (siehe dazu auch Abschnitt 5.20.5) werden die Datei eingelesen und die Entitäten zwischengespeichert.

⟨*RALChecker main function body*⟩+≡
```
1  print("Try to load " + filename);
2  IfcStepReader reader = new IfcStepReader();
3  IFC4X1Model model = reader.read(filename);
```

Eine Trassierung wird im IFC 4.1-Schema auf Basis der Klasse IfcAlignment beschrieben. Daher suchen wir zunächst alle Entitäten, die von diesem Typ sind. Haben wir eine Instanz vom Typ IfcAlignment gefunden, so erzeugen wir eine Instanz der Klasse RALChecker, deren Konstruktor als Parameter eine Variable vom Typ IfcAlignment erwartet. Zudem bietet die Klasse RALChecker eine öffentliche Methode namens check an, die die Prüfung entsprechend durchführen soll.

⟨*RALChecker main function body*⟩+≡
```
1  for(int i = 0; i < model.getEntitiesCount(); i++) {
2      IFC4X1Entity e = model.getEntityByIndex(i);
3
4      if(e instanceof IfcAlignment) {
5          IfcAlignment a = cast<IfcAlignment>(e);
6
7          RALChecker checker = new RALChecker(a);
8          checker.check();
9      }
10 }
```

8.3.2.2 Die Klasse RALChecker

Die Klasse RALChecker besitzt neben einem Konstruktor auch noch zwei Membervariablen. Diese speichern das zugewiesene Alignment (alignment_) und einen Wert, der die aktuelle Stationierung (currentStation_) beschreibt. Zunächst ist die aktuelle Stationierung unbekannt, daher wird diese vorerst auf den Wert 0 initialisiert.

⟨*RALChecker.ifcpl*⟩+≡
```
1  class RALChecker {
2      public RALChecker(IfcAlignment a) {
3          alignment_ = a;
4      }
5
6      ⟨RALChecker class body 184⟩
```

```
7
8     private double currentStation_ = 0;
9     private IfcAlignment alignment_;
10 };
```

Die Methode check der Klasse RALChecker ruft eine visit-Methode auf.

⟨*RALChecker class body*⟩≡

```
1 public void check() {
2     visit(alignment_);
3 }
```

Die visit-Methode hat das Ziel, sich, ausgehend von der IfcAlignment-Klasse, zu den einzelnen IfcCircularArcSegement2D-Instanzen zu hangeln (siehe dazu Abbildung 8.29). Dabei geht die Methode davon aus, dass die Achse des Alignments mithilfe einer Instanz der Klasse IfcAlignmentCurve beschrieben wurde. Sollte dies nicht der Fall sein, wird dem Benutzer eine entsprechende Meldung angezeigt. Die Überprüfung, ob das Attribut Axis belegt ist (`a.Axis != null`), kann entfallen, da dieses Attribut ein Pflichtattribut ist. Ist der Typ des Attributs Axis vom Typ IfcAlignmentCurve, so wird das entsprechende Objekt mithifle einer überladenen visit-Methode besucht.

⟨*RALChecker class body*⟩+≡

```
1 private void visit(IfcAlignment a) {
2     if(a.Axis instanceof IfcAlignmentCurve) {
3         IfcAlignmentCurve ac = cast<IfcAlignmentCurve>(a.Axis);
4         visit(ac);
5     }
6     else {
7         print("Unkown alignment curve type.");
8     }
9 }
```

Über die IfcAlignmentCurve-Instanz kann man zu der Membervariablen Horizontal navigieren. Diese ist vom Typ IfcAlignment2DHorizontal und besitzt wiederum das Attribut Segments. Das Attribut stellt eine Liste von IfcAlignment2DHorizontalSegment-Instanzen bereit. Diese werden ebenfalls wieder mittels einer visit-Methode besucht. Bevor über diese Liste iteriert wird, wird jedoch erst noch die Membervariable currentStation_ mit dem Wert des Attributs StartDistAlong initialisiert, das die Stationierung am Startpunkt des Alignments speichert.

⟨*RALChecker class body*⟩+≡

```
1 private void visit(IfcAlignmentCurve ac) {
2     IfcAlignment2DHorizontal ha = ac.Horizontal;
3     currentStation_ = ha.StartDistAlong.getValue();
4     for(int i = 0; i < ha.Segments.count(); i++) {
5         visit(ha.Segments[i]);
6     }
7 }
```

Ist man schließlich bei der IfcAlignment2DHorizontalSegment-Instanz angelangt, kann man jetzt zum Attribut CurveGeometry navigieren und dort prüfen, ob man

es mit einem IfcCircularArcSegment2D-Element zu tun hat. Falls dem so ist, wird die visit-Methode für das Kreissegement (IfcCircularArcSegment2D) aufgerufen. Im Anschluss wird der aktuelle Stationierungswert currentStation_ aktualisiert. Um dies zu erreichen, wird der Wert des Attributs SegmentLength auf den aktuellen Stationierungswert aufaddiert.

⟨*RALChecker class body*⟩+≡

```
1  private void visit(IfcAlignment2DHorizontalSegment hs) {
2      IfcCurveSegment2D cs = hs.CurveGeometry;
3
4      if(cs instanceof IfcCircularArcSegment2D ) {
5          IfcCircularArcSegment2D arcSegment =
6              cast<IfcCircularArcSegment2D>(cs);
7          visit(arcSegment);
8      }
9
10     currentStation_ += cs.SegmentLength.getValue();
```

Nun kann die eigentliche Regelprüfung erfolgen. Da wir implizit von der Entwurfsklasse EKL 1 ausgehen, muss der Radius eines Kreisbogens im Lageplan größer als $500\,m$ sein und die Länge des Kreisbogens mindestens $70\,m$ betragen. Falls zumindest eine der Regeln verletzt wird, wird der Name des Alignment ausgegeben, bei dem die Regel verletzt wird (insofern dieser vorhanden ist), zusammen mit dem Stationierungswert des Startpunktes des regelverletzenden Kreisbogenelements.

⟨*RALChecker class body*⟩+≡

```
1  private void visit(IfcCircularArcSegment2D arcSegment) {
2      bool reportError = false;
3
4      if(arcSegment.Radius.getValue() < 500) {
5          print("Rule (Radius >= 500) is violated!");
6          reportError = true;
7      }
8
9      if(arcSegment.SegmentLength.getValue() < 70) {
10         print("Rule (Length >= 70) is violated!");
11         reportError = true;
12     }
13
14     if(reportError) {
15         if(alignment_.Name != null) {
16             print("Alignment name: " + alignment_.Name.getValue());
17         }
18
19         print("Station: " + currentStation_);
20     }
21 }
```

Der vollständige Quelltext zum RALChecker befindet sich im Anhang B.3.4. Abbildung 8.30 zeigt eine mögliche Ausgabe des RALCheckers.

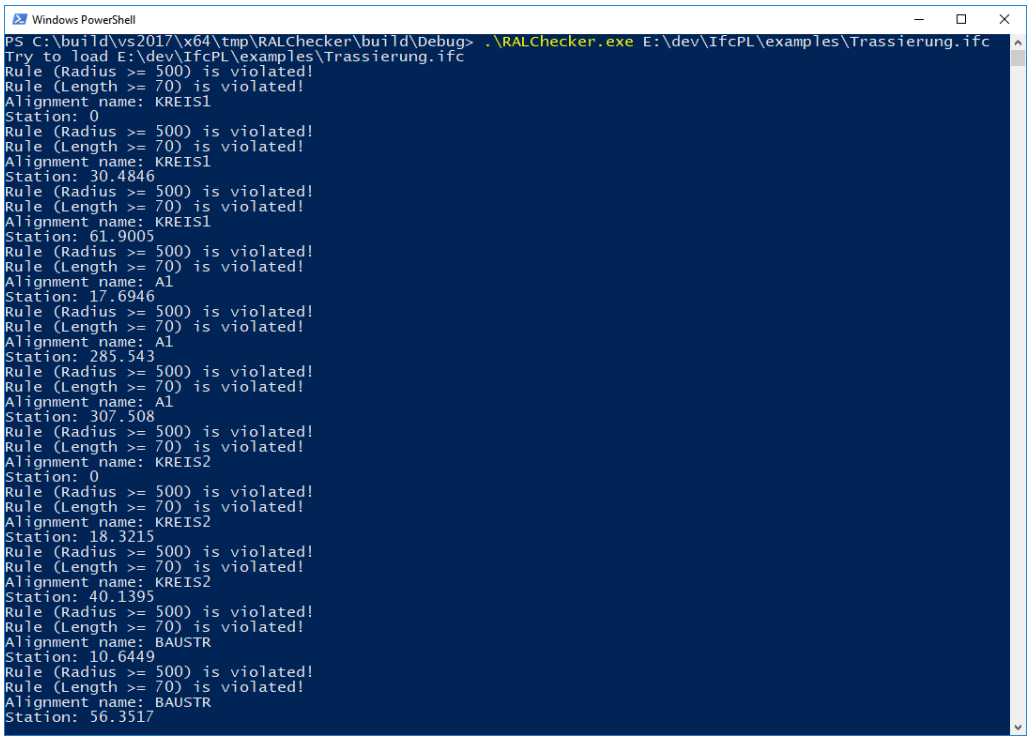


Abbildung 8.30: Mögliche Ausgabe das RALCheckers

8.3.2.3 Erweiterung der Klasse RALChecker

Mit einem geringen Mehraufwand könnte man einen Aufzählungsdatentyp für die Entwurfsklasse einführen und, abhängig von der Entwurfsklasse, die entsprechenden Radien überprüfen.

Zudem müsste man auch noch die verwendeten Einheiten des Projekts überprüfen, da man nicht immer davon ausgehen kann, dass alle Einheiten in Meter gespeichert sind. Diese Überprüfung könnte etwa so aussehen:

```
1  if(e instanceof IfcProject) {
2      IfcProject project = cast<IfcProject>(e);
3
4      IfcUnitAssignment unitAssignment = project->UnitsInContext;
5
6      for(int i = 0, i < unitAssignment.Units.count(); ++i) {
7          if(unitAssignment.Units[i] instanceof IfcSIUnit) {
8              IfcSIUnit unit = unitAssignment.Units[i];
9              if(unit.UnitType == IfcUnitEnum.LENGTHUNIT) {
10                 print("Assuming meter as length unit");
11                 assert(unit.Prefix == null);
12                 assert(unit.Name == IfcSIUnitName.METRE);
13             }
14         }
15     }
16 }
```

Anstatt den Programmablauf abzubrechen, wenn die entsprechenden Bedingungen nicht erfüllt sind, könnte man auch die Einheiten entsprechend umrechnen.

Abbildung 8.31 zeigt den beschriebenen Ansatz im Überblick. Die Richtlinien werden durch einen Programmierer interpretiert und in ein IFC-PL-Programm übersetzt. Das IFC-PL-Programm selbst wird dann mit dem IFC-PL-Übersetzer in eine ausführbare Datei (.exe) übersetzt. Dieses Programm erwartet als Kommandozeilenparameter eine IFC 4.1-STEP-Datei, welche Trassierungsdaten enthält. In einem hier nicht umgesetzten optionalen Vorverarbeitungsschritt könnten die Daten auf Meter umgerechnet werden, falls sie noch nicht in dieser Einheit vorliegen. Im Programmablauf werden die vordefinierten Regeln überprüft und bei einer Regelabweichung eine Meldung an den Benutzer ausgegeben. Versierte BIM-Anwender mit Programmiererfahrung können die Regeln im Detail untersuchen und verifizieren. Nicht-Programmierer können das Programm RALChecker.exe einfach als Black-Box verwenden und so gegebenenfalls Regelverletzungen identifizieren.

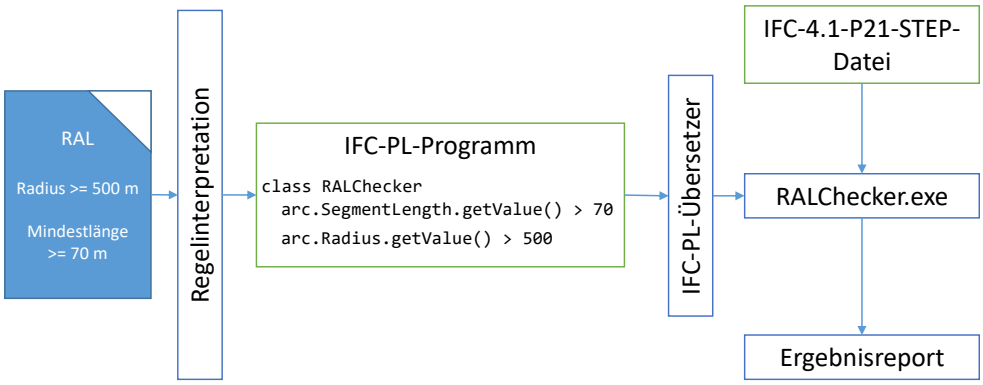


Abbildung 8.31: Beschriebener Ansatz im Überblick

Auch andere Regeln der RAL lassen sich auf ähnliche Weise mit der IFC-PL erfassen. Beispielsweise schreibt die RAL vor: „... die Länge von Geraden [soll] auf max L = 1.500 m begrenzt werden.“. Diese lässt sich ebenfalls durch eine Ergänzung des gezeigten Beispiels relativ einfach umsetzen, wie im Folgenden dargestellt ist:

```
1 private void visit(IfcLineSegment2D lineSegment) {
2     if(lineSegment.SegmentLength.getValue() > 1500) {
3         print("Rule is violated!");
4     }
5 }
```

Ein anderes Beispiel für eine Richtlinie der RAL ist die Vorgabe von Kurvenmindestradien bei der Elementfolge Gerade - Klothoide - Kreisbogen. Tabelle 8.2 gibt hierzu einen kleinen Überblick (Diese gilt nur für bestimmte Straßenkategorien. Siehe dazu auch (FGSV 2013)).

Die Prüfung dieser Regel könnte in etwa so aussehen:

Länge L[m] der Geraden	**min R [m] des Kreisbogens**
$L \geq 300\,m$	$\min R > 400\,m$
$L < 300\,m$	$\min R > L$

Tabelle 8.2: Kurvenmindestradien bei der Elementfolge Gerade - Klothoide - Kreisbogen

```
1  private bool checkCurveMinRadius() {
2     // consider always three successive segments
3     for(i = 0; i <= alignment.Segments.count()-3; i++) {
4        if(alignment.Segments[i+0] instanceof IfcLineSegment2D &&
5           alignment.Segments[i+1] instanceof IfcTransitionCurveSegment2D &&
6           alignment.Segments[i+2] instanceof IfcCircularArcSegment2D) {
7            line = alignment.Segments[i+0];
8            arc = alignment.Segments[i+2];
9
10           if(line.SegmentLength.getValue() >= 300.0) {
11              if(arc.Radius <= 400.0)
12                 return false;
13              }
14           else {
15              if(arc.Radius.getValue() < line.SegmentLength.getValue())
16                 return false;
17           }
18        }
19    }
20
21    return true;
22 }
```

8.3.2.4 Schnittstelle für die Integration in Anwendungen

Anstatt die Regelüberprüfung im Standalone-Modus durchzuführen, könnte diese auch über eine vergebene Schnittstelle in verschiedene Anwendungen integriert werden. Eine mögliche Realisierung für den gezeigten Anwendungsfall könnte die Schnittstelle IChecker sein:

```
1  interface IRuleViolation {
2     double getStation();
3     string getDescription();
4  }
5
6  interface ICheckResult {
7     int getRuleViolationCount();
8     IRuleViolation getRuleViolationByIndex(const int index);
9  }
10
11 interface IChecker
12 {
13    ICheckResult check();
14 }
```

Das Interface IChecker könnte durch ein IFC-PL-Programm realisiert werden, das entsprechend in den Datenaustausch integriert werden könnte. ICheckResult stellt dabei die Ergebnisse einer Regelprüfung mittels der Methode check dar. Dabei verwaltet eine Implementierung der ICheckResult-Schnittstelle eine beliebige Anzahl von Regelverletzungen IRuleViolation. Für eine Regelverletzung kann eine Beschreibung (getDescription) und die zugehörige Stationierung ermittelt werden (getStation). Dieser Ansatz würde es einer Anwendung auch ermöglichen, eine Regelverletzung grafisch referenziert darzustellen (siehe Abbildung 8.32).

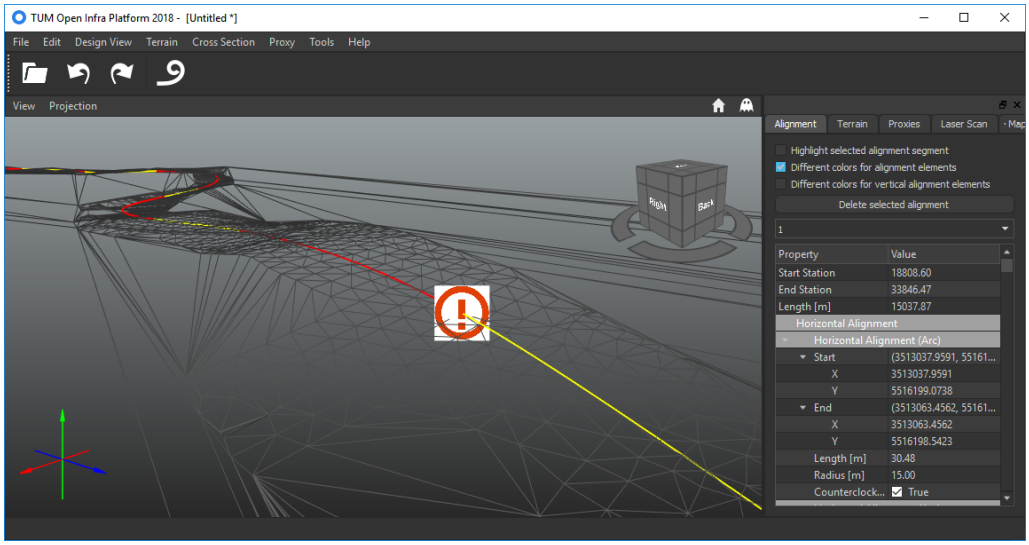


Abbildung 8.32: Mögliche Visualisierung einer RAL-Regelverletzung mithilfe eines roten Warnungssymbols

8.3.3 Weitere Anwendungen in der Regelprüfung

Es wurde exemplarisch gezeigt, wie sich Regeln der RAL mithilfe der IFC-PL abbilden lassen und die IFC-PL auch für den gezeigten Anwendungsfall eingesetzt werden kann. Ähnlich kann man auch noch weitere Regeln der RAL abbilden und herstellerneutral austauschen. Darüber hinaus könnte man auch länder- bzw. projektspezifische oder firmeninterne Regeln abbilden. Das Prinzip ist theoretisch auch auf andere Bauwerksdatenmodelle übertragbar, z. B. auf den OKSTRA-Standard (siehe dazu auch Abschnitt 5.20.7).

Die Digitalisierung von Bauplänen und -prozessen durch Building Information Modeling versetzt Baubeteiligte wie etwa Prüfingenieure in eine klassen- und objektorientierte Welt der Bauwerksinformationsmodelle. Der hier vorgestellte Whitebox-getriebene und modellbasierte Prüfungsansatz setzt Wissen über das zugrundeliegende Bauwerksdatenmodell voraus und ermöglicht es, auf Basis von IFC-PL Regelprüfungen zu formulieren und diese abzuprüfen. IFC-PL ist sicherlich nicht für Anwender ohne Programmierkenntnisse zugänglich und zielt auch gar nicht auf diese Anwendergruppe der Nicht-Programmierer ab. Jedoch bietet er Personen mit Programmierkenntnissen der C++-Sprachfamilie die Möglichkeit, eng verzahnt an einem IFC-basierten Bauwerksmodell einfache Regeln zu prüfen,

und er ist darüber hinaus herstellerneutral. Einfach wiederkehrende Regelprüfungen können auf diese Art und Weise automatisiert werden.

Es ist auch denkbar, auf Basis der IFC-PL einen Codegenerator zu entwickeln. Dabei wäre es vorstellbar, dass ein Benutzer auf Basis von Satzbausteinen einfach Sätze zusammenstellt, die dann gleichzeitig in IFC-PL-basierte Regeln für das Bauwerksmodell übersetzt werden (siehe Abbildung 8.33).

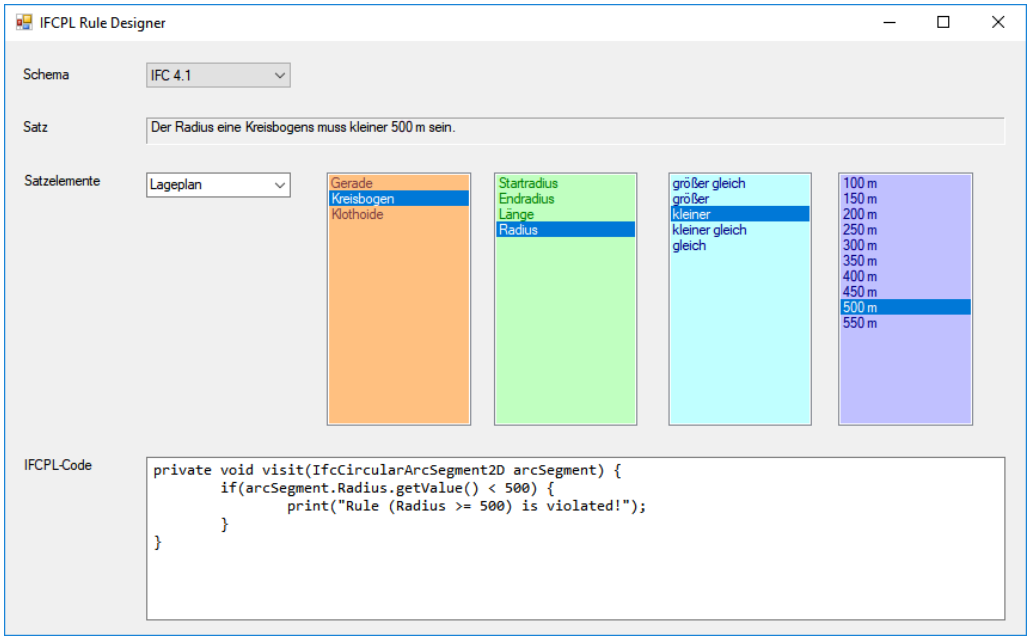


Abbildung 8.33: Prototypischer Codegenerator, der auf Basis vorgefertigter Satzbausteine IFC-PL-Prüfcode generiert

Der vorgestellte Ansatz ist auch übertragbar auf Richtlinien des Hochbaus. Beispielsweise wird in den Technischen Regeln für Arbeitsstätten (ASR) für Fluchtwege gefordert, dass diese, abhängig von der Anzahl der Personen, bestimmte lichte Breiten einzuhalten haben. Für einen Fluchtweg, der für 200 Personen ausgelegt ist, muss beispielsweise eine lichte Breite von $1,20\,m$ eingehalten werden. Dies gilt natürlich auch für Engstellen wie Türen. Ein IFC-PL-Programm, das alle Türelemente (IfcDoor) in einem IFC-basierten Bauwerksdatenmodell ermittelt und die Breite dieser Türen auf die Mindestanforderungen überprüft, ist relativ trivial realisierbar. Schwieriger wird es hingegen bei der Bestimmung, ob bestimmte Gänge die Mindestbreite erfüllen, weil diese unter Umständen nur aus dem räumlichen Wissen von verschiedenen Bauteilen ermittelbar und nicht direkt aus dem Modell ablesbar ist. Hier sind Ansätze wie (Preidel & Borrmann, 2016), die räumliche Abfragen ermöglichen, vielversprechender. Dennoch wäre es auch vorstellbar, die Standardbibliothek der IFC-PL mit einem Modul für räumliche Abfragen auszustatten.

8.4 Zusammenfassung

In diesem Kapitel wurden drei verschiedene Anwendungsfälle für den Einsatz von IFC-PL beschrieben.

Im Beispiel der Übergangskurven aus Abschnitt 8.1 wurde eine generische Schnittstelle verwendet, die für verschiedene Übergangskurven implementiert werden kann. Durch diesen Ansatz ergibt sich eine Reihe von Vorteilen. U. a. ermöglicht dieses Vorgehen die Einführung neuer Übergangskurventypen zur Laufzeit, ohne dass eine erneute Standardisierung nötig ist. Außerdem wird dadurch ein Softwarehersteller nicht gezwungen, jede Variante einer Übergangskurve zu implementieren, jedoch werden fremde Implementierungen implizit über die vereinbarte Schnittstelle unterstützt.

Mit der beschriebenen Instanz-Level-Metrik (siehe Abschnitt 2.5) wird die Komplexität eines BIM-Modells durch die Anzahl der in einer Instanzdatei verwendeten Entitäten definiert. Am Beispiel der flexiblen Unterstützung von parametrischen Profildefinitionen wurde gezeigt, dass bei diesem Ansatz neue Profildefinitionen eingeführt werden können, ohne die Komplexität des Datenmodells zu erhöhen. Insbesondere das Hinzufügen neuer Profildefinitionen hat keinen Einfluss auf die Modellkomplexität, da keine neuen Entitäten im Schema und in der Instanzdatei eingeführt werden. In einem konventionellen Modell ohne dynamische Erweiterungsmöglichkeit muss das Schema modifiziert und erweitern werden, um neue Profildefinitionen zu unterstützen. Dadurch wird letztlich die Instanz-Level-Metrik-Komplexität erhöht. Der gezeigte Ansatz kann hier helfen, mit der zunehmenden Komplexität von Bauwerksdatenmodellen umzugehen.

Am Beispiel des rechtwinkligen Kastenwiderlagers wurde gezeigt, wie konstruktives Wissen in Form von IFC-PL-Programmen integriert und zwischen verschiedenen Anwendungen ausgetauscht werden kann. Darüber hinaus wurde dadurch eine Möglichkeit aufgezeigt, wie intelligente BIM-Objekte mithilfe der IFC-PL realisiert werden können.

Im letzten Teil wurde auf die Prüfung von Normen und Richtlinien eingegangen und exemplarisch an Beispielen der Richtlinien für die Anlage von Landstraßen (RAL) aufgezeigt, wie sich Richtlinien mithilfe der IFC-PL überprüfen lassen.

Kapitel 9

Alternative Ansätze

In diesem Kapitel werden alternative Ansätze und Konzepte zur IFC-PL vorgestellt und kurz skizziert. Diese unterscheiden sich in ihrer technologischen Umsetzung, bieten jedoch ähnliche Vorteile wie beispielsweise die flexible Erweiterbarkeit eines Bauwerksmodells.

9.1 Ein XML-basierter Ansatz

Der IFC-PL-Ansatz stellt nicht die einzige Möglichkeit dar, das in Kapitel 4 vorgestellte Konzept umzusetzen. Ein anderer Realisierungsweg kann beispielsweise auf Basis von XML-Technologien beschritten werden. Dabei wird anstatt der IFC-PL die XML Query Language (XQuery) verwendet. XQuery ist eine wohldefinierte und standardisierte Sprache, zu der zahlreiche Ressourcen wie beispielsweise Bücher oder Dokumentationen verfügbar sind (Brundage, 2004; Walmsley, 2015; W3C, 2014). Außerdem gibt es viele Experten, die diese Sprache beherrschen. Daneben existieren bereits eine Reihe von Ausführungsumgebungen (Implementierungen), die XQuery-Anfragen verarbeiten können.

Im Folgenden soll anhand eines Beispiels skizziert werden, wie sich ein XML-basierter Ansatz umsetzen lässt. Als Beispiel soll hier der Anwendungsfall der Übergangskurven, der in Abschnitt 8.1 beschrieben wurde, aufgegriffen werden. Bei der Implementierung des Interfaces IArbitraryTransitionCurve aus dem Fallbeispiel der Übergangskurven muss die Methode getPosition implementiert werden. Für eine Übergangskurve müssen dabei entsprechend die x- und y-Koordinate berechnet werden. Bei der Realisierung einer Klothoide mittels eines XQuery-Programms können beispielsweise entsprechend die Fresnel-Integrale berechnet werden. Für die Berechnung der x-Koordinate der Klothoide (bezogen auf ein lokales Koordinatensystem) ergibt sich dabei folgender XQuery-Ausdruck:

```
1 declare function local:computeX(
2  $L as xs:double, $A as xs:double, $iterations as xs:integer
3 ) as xs:double {
4  fn:sum(
```

```
 5   for $i in (1 to $iterations)
 6   let $sign := $i mod -1
 7   let $L_exponent := 5+ ($i - 1) * 4
 8   let $A_exponent := $i * 4
 9   let $factor := local:factorial(2 * $i) * local:pow(2, 2 * $i) * (5+($i - 1)*4) + 3
10   let $tmp := local:pow($A, $A_exponent)
11   let $term := $sign * local:pow($L, $L_exponent) div ($factor * $tmp)
12   return $term
13   ) + $L
14  };
```

XQuery-Anweisungen müssen dem FLWOR-Schema folgen. Ein FLWOR-Ausdruck[1] setzt sich aus For-, Let-, Where-, OrderBy- und Return-Klauseln zusammen, welche genau in dieser Reihenfolge erscheinen müssen. XQuery (in der Version 3.0) ist Turing-vollständig, was vereinfacht bedeutet, dass alles, was in einer Programmiersprache wie der IFC-PL ausgedrückt werden kann, auch mittels XQuery ausgedrückt bzw. berechnet werden kann. Der eigentliche Zweck bzw. Entstehungsgrund von XQuery war die Notwendigkeit, eine Abfragesprache für XML-Dokumente bereitzustellen. Neben funktionalen motivierten Sprachelementen besitzt XQuery auch einige deklarative Elemente.

Das hier beschriebene Klothoiden-Programm hängt von den folgenden Eigenschaften ab: der Startposition, der Startrichtung, der Startkrümmung, der Orientierung, dem Eingangswert (isEntry), der Bogenlänge sowie der Klothoidenkonstante der durch das Programm beschriebenen Klothoide. Die gleichen Parameter wurden auch für die Beschreibung von Klothoiden in IFC-Alignment 1.0 (Amann *et al.*, 2014) verwendet. Für weitere Informationen zu diesen Parametern steht die Dokumentation der IFC Alignment 1.0 bereit (buildingSMART, 2018d). Die Werte dieser Klothoidenparameter werden in einem entsprechenden ifcXML-Dokument gespeichert. Dieses ist auszugsweise in Abbildung 9.1 abgebildet.

Um mit XQuery auf diese Werte zugreifen zu können, kann eine XPath-Abfrage (Simpson, 2002) verwendet werden, wie im folgenden Listing dargestellt ist:

```
1  declare variable $startPositionX_ :=
2  IfcTransitionCurve/IfcPropertySingleValue[Name="startPosition_"]/NominalValue/
       IfcLengthMeasure-wrapper/data();
```

Im Listing wird die XPath-Verwendung innerhalb einer XQuery gezeigt, um auf die Start-x-Position einer Klothoide zuzugreifen. Auf diese Weise kann ein XQuery-Programm definiert werden, das den entsprechenden Positionswert für einen bestimmten Stationierungswert auf einer Klothoide berechnet. Auf diese Weise können auch Programme definiert werden, die die Länge und Krümmung einer Klothoide berechnen. Dieser Ansatz erlaubt es, eigene Attribute für die Verwendung in XQuery-Programmen innerhalb eines Property-Sets zu definieren.

Im Gegensatz zum IFC-PL-Ansatz weist der XML-basierte Ansatz u. a. folgende Unterschiede auf:

[1] FLWOR wird als *flower* ausgesprochen

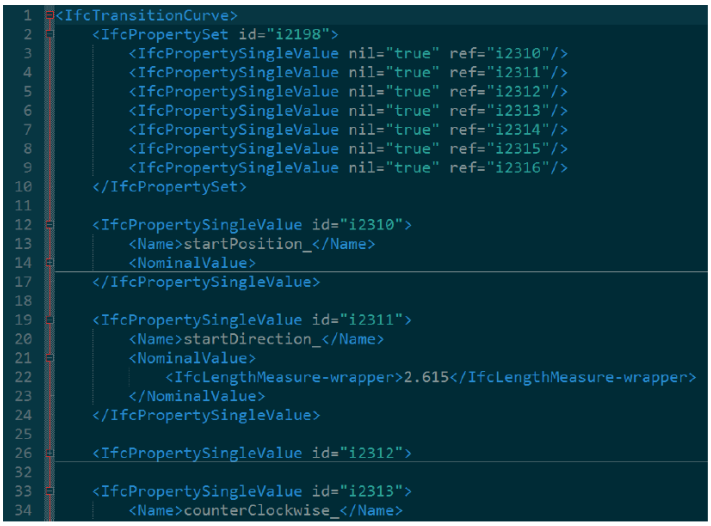


Abbildung 9.1: Screenshot eines ifcXML4-Dokuments

- Die STEP-P21 Instanzdateien werden durch ifcXML-Instanzdateien ersetzt. Dies ist nötig, da mittels XPath auf Attribute des IFC-Property-Sets zugegriffen werden muss.
- Programme werden als XQuery-Ausdrücke formuliert, anstatt diese in IFC-PL zu formulieren. XQuery-Programme werden in der Form von *.xq-Dateien ausgetauscht.
- Die IFC-PL-Laufzeitumgebung wird durch einen XQuery-Interpreter ersetzt. Ein XQuery-Interpreter wird standardmäßig z. B. vom .NET Framework oder der Java Platform Standard Edition angeboten.

XML und XQuery sind sehr verbreitet und werden von vielen Programmierbibliotheken für verschiedene Programmiersprachen unterstützt. Im Vergleich dazu hat der EXPRESS-Softwarestack nur eine sehr geringe Verbreitung. Grundsätzlich findet man mehr Ansprechpartner und Ressourcen zum Thema XML als zum Thema EXPRESS. Dennoch hat der XQuery-basierte Ansatz auch einige Nachteile: Umständliche Zugriffsmuster (via XPath) und die Notwendigkeit, Programme immer in der Form einer FLWOR-Expression formulieren zu müssen, machen die Benutzung von XQuery umständlich. Dies ist auch nicht sonderlich überraschend, da die Sprache mit dem Hauptaugenmerk auf Abfragen für XML-Dokumente erstellt wurde und die hier beschriebenen Zwecke nicht als Leitmotive beim Design der Sprache berücksichtig worden sind. Zudem passt der XQuery-Ansatz nicht gut in die traditionelle STEP/CTE-Welt, und es ist unklar, ob auf lange Sicht der IFC-Standard komplett auf XML umgestellt wird. Hierbei ist auch anzumerken, dass ifcXML derzeit keine Pendants von EXPRESS-Funktionen, Regeln oder Where-Clauses in der XML-Welt besitzt. Um auch diese fehlenden Elemente zu unterstützen, ist es denkbar, diese mithilfe von XQuery abzubilden. Ob sich für diesen Zweck tatsachlich XQuery eignet, ist jedoch noch eine offene Fragestellung, die im Rahmen von weiterführenden Arbeiten in Bereich von XML-Technologien im Zusammenhang mit IFC geklärt werden müsste.

9.2 Linked-Data

IFC-PL ermöglich im Rahmen bestimmter Grenzen die Erweiterung eines Bauwerksdatenmodells. Jedoch ist es nicht zwangsläufig zielführend, ein Bauwerksdatenmodell immer weiter mit mehr Wissen anzureichen. Für bestimmte Domänen gibt es oft unterschiedliche Datenmodelle und der Aufwand, ein Datenmodell auf ein anderes zu migrieren, ist oft zu hoch. Hier bietet der Linked-Data-Ansatz eine mögliche Lösung. Anstatt ein großes Datenmodell zu schaffen, das alle Aspekte umfasst, ist es auf Basis von Linked-Data-Technologien (Wood *et al.*, 2014) möglich, verschiedene Datenmodelle miteinander zu verknüpfen.

Beispielsweise bilden OKSTRA 2.017 und IFC 4.1 ähnliche Fachobjekte wie z. B. Entwurfsobjekte einer Trassierung ab, unterscheiden sich jedoch stark in großen Teilen ihrer Datenmodelle. So besitzt beispielsweise IFC 4.1 weder Entitäten zur Beschreibung einzelner Unfälle oder zur Darstellung von streckenbezogenen Unfallkenngrößen noch die Möglichkeit, Objektarten aus der Ökologie wie etwa Baumbestände, die für das Straßen- und Verkehrswesen relevant sind, abzubilden. OKSTRA hingegen besitzt keine Möglichkeiten, Brückenbauwerke geometrisch detailliert zu beschreiben, die IFC 4.1 jedoch abbilden kann (siehe Abbildung 9.2).

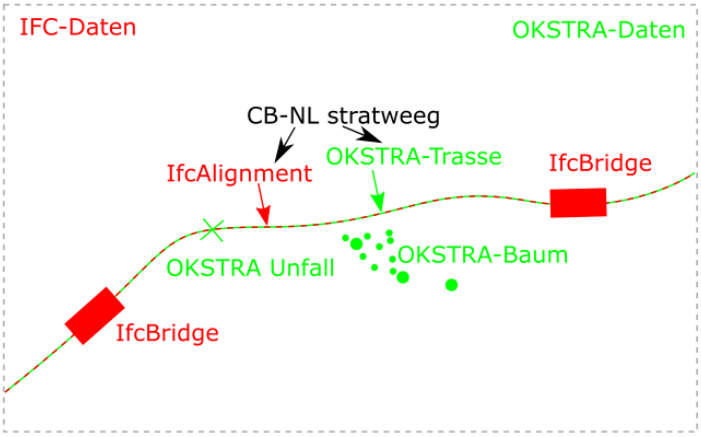


Abbildung 9.2: OKSTRA und IFC besitzen eine Teilmenge ähnlcher Entitäten, unterscheiden sich jedoch auch in großen Teilen (Rot = IFC-Daten, Grün = OKSTRA-Daten). Konzeptbibliotheken wie die CB-NL beschreiben allgemeine Konzepte, die teilweise in beiden Datenmodellen zu finden sind.

Damit eine Fachapplikation auf beide Datenbestände zugreifen kann, gibt es prinzipiell verschiedene Vorgehensweisen.

Beispielsweise können beide Datenmodelle einfach durch eine Fachapplikation unterstützt werden. Dies ist z. B. der Fall in der Open Infra Platform. Bei dieser Vorgehensweise muss eine Anbindung für das OKSTRA- und das IFC-Datenmodell geschaffen werden. Dabei gibt es technologische Unterschiede bei beiden Anbindungen. OKSTRA 2.017 ist XML-basiert, IFC ist EXPRESS-basiert. Für die OKSTRA-Anbindung wird die OKLABI-Bibliothek bereitgestellt, die von der

OIP genutzt wird, für IFC wird ein Early-Binding-Ansatz verwendet. Das Handling der Daten ist in beiden Fällen unterschiedlich.

Eine andere Variante, die beide Datenmodelle auf eine gemeinsame technologische Basis stellt, ist ein Linked-Data-Ansatz. Dabei werden die Daten des OKSTRA- und des IFC-Modells einer Fachapplikation jeweils durch einen SPARQL-Endpoint bereitgestellt. SPARQL ist eine Abfragesprache für RDF (DuCharme, 2013). Ein SPARQL-Endpoint ist durch eine HTTP-URL erreichbar, die SPARQL-Abfrage akzeptiert und die Ergebnisse einer an den Endpoint gesendeten SPARQL-Abfrage zurückgibt. Im Regelfall unterstütz der SPARQL-Endpoint dabei eine Vielzahl verschiedener Serialisierungen wie z. B. Turtle oder RDF XML. Damit dies funktionieren kann, bedarf es folgender Grundvoraussetzungen:

- Es steht eine Ontologie für OKSTRA und IFC bereit.
- Es gibt ein Werkzeug, das OKSTRA-XML bzw. IFC-EXPRESS-Daten in entsprechende RDF-Daten umwandeln kann.
- Es werden Server mit entsprechenden SPARQL-Endpoints eingerichtet.
- Die Fachapplikation muss eine Bibliothek einbinden, die SPARQL-Abfragen erlaubt.

Auf der Ebene der Fachapplikation kann dadurch auf die Anbindung der OKLA-BI bzw. eines IFC-Early-Bindings verzichtet werden. Es gibt nur noch SPARQL-Anfragen auf Ebene der Fachapplikation und damit nur noch eine technologische Vorgehensweise, um Abfragen an beide Modelle zu stellen. Abbildung 9.3 veranschaulicht dieses Vorgehen.

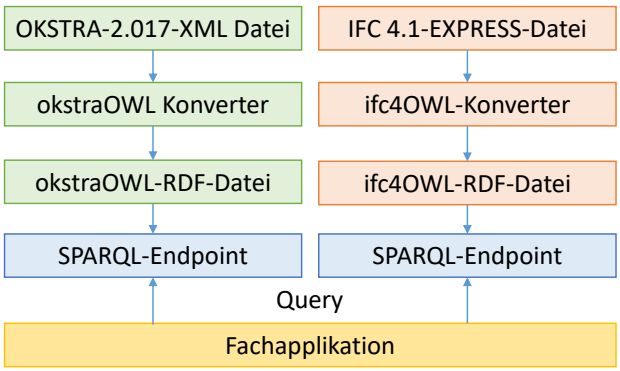


Abbildung 9.3: SPARQL-Anfragen auf Ebene der Fachapplikation

Möchte man beispielsweise durch einen Linked-Data-Ansatz zu einer in IFC abgespeicherten Trassierung alle Unfallorte ermitteln, die in einem OKSTRA-Datenmodell gespeichert sind, so kann man wie folgt vorgehen: Zunächst konvertiert man die IFC- und OKSTRA-Daten in RDF-Daten. Die OIP bietet hier entsprechende Funktionalitäten für den Export von RDF-Daten für eine Teilmenge von Klassen und Attributen beider Datenformate an. Die exportierten RDF-Daten müssen dann in einen Triplestore importiert werden. Ein Triplestore ist eine Datenbank, die auf Verwaltung RDF-Triple optimiert ist.

Als RDF-Datenbank kann beispielsweise die Software BrightstarDB verwendet werden (siehe Abbildung 9.4). In diese werden beide Datensätze (oksatraOWL- und ifcOWL-Instanz-Daten) geladen. Dabei werden beide Datensätze in einem gemeinsamen Store vereinigt. Eine andere Vorgehensweise wäre, jeweils zwei unterschiedliche Stores zu verwenden, die über zwei unterschiedliche SPARQL-Endpoints angeboten werden. Dieser Ansatz wird am Ende dieses Abschnittes noch kurz erläutert.

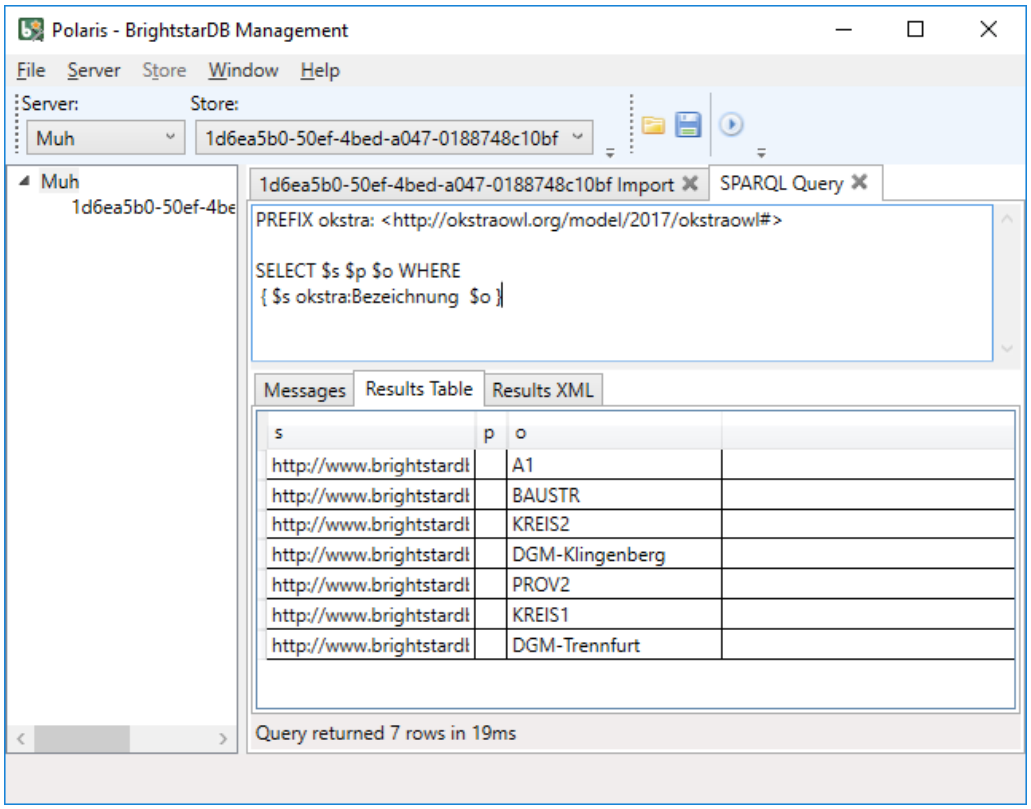


Abbildung 9.4: Die Software BrightstarDB stellt eine Infrastruktur für Triplestores bereit. Nach dem Import der OKSTRA-RDF-Daten in den Triplestore können ebenfalls SPARQL-Abfragen in BrightstarDB getätigt werden, wie in der Abbildung zu sehen ist. In der Beispielabfrage wurde nach den Bezeichnungen aller Objekte angefragt. Das zugrundeliegende Modell beschreibt fünf Trassierungen (mit den Namen A1, BAUSTR, KREIS2, PROV2, KREIS1) und zwei Geländemodelle (DGM-Klingenber, DGM-Trennfurt).

Folgende Abfrage ermittelt zu allen IFC-Alignment-Elementen die entsprechenden OKSTRA-Unfallpositionen:

```
1  PREFIX ifc: <http://www.buildingsmart-tech.org/ifcowl/IFC4x1#>
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX okstra: <http://okstraowl.org/model/2017/okstraowl#>
4
5  SELECT ?UnfallPositionen WHERE
6  {
7      ?s rdf:type ifc:IfcAlignment .
8      ?s ifc:name_IfcRoot ?Strassenname .
9
10     ?s2 okstra:hat_Strassenbezeichnung____Strasse ?Strassenname .
11     ?s2 okstra:von_Unfallort____Strasse ?UnfallObjekt .
12     ?UnfallObjekt okstra:Punktgeometrie____Angaben_zum_Unfallort ?UnfallPositionen
13 }
```

Eine entsprechende Fachapplikation kann auf diese Weise für IFC-Trassierungsdaten die entsprechenden OKSTRA-Unfall-Daten ermitteln, ohne das Linked-Data-Umfeld verlassen zu müssen. Es müssen keine EXPRESS-spezifischen Early-Bindings verwendet werden, es ist keine Auseinandersetzung mit der OKLABI nötig und es können einfach weitere Ontologien eingebunden werden. Eine mögliche Visualisierung der SPARQL-Abfrage ist in Abbildung 9.5 dargestellt.

Abbildung 9.5: Mögliche Visualisierung aller Unfallorte einer Fachapplikation auf Basis von ifcOWL und okstraOWL

Bei Bäumen, Verkehrsanlagen und ähnlichen Objekten ist ein ähnliches Vorgehen möglich.

Möchte man die Daten nicht in einem gemeinsamen Store vereinigen, kann man auch zwei getrennte Server dafür benutzen. In früheren Versionen von SPARQL war dies nicht möglich. Der gängige Workaround sah vor, beide Triple-Stores downzuloaden und diese entsprechend in einer Datei zu kombinieren. Seit Veröffentlichung der SPARQL-Version 1.1 gibt es das SERVICE-Keyword, das es erlaubt, eine Query über mehrere Endpoints hinweg durchzuführen. Im Folgenden ist eine Abfrage dargestellt, die mehrere Endpoints nutzt:

```
1 PREFIX bibleontology: <http://bibleontology.com/resource/>
2 PREFIX dbo: <http://dbpedia.org/ontology/>
3 PREFIX owl: <http://www.w3.org/2002/07/owl#>
4 SELECT ?art ?abstract
5 WHERE {
6 SERVICE <http://bibleontology.com/sparql/> { bibleontology:Ezra owl:sameAs ?art . }
7 SERVICE <http://dbpedia.org/sparql> { ?art dbo:abstract ?abstract . } }
```

Ein Linked-Data-Ansatz schließt den IFC-PL-Ansatz nicht aus, sondern beide Ansätze können sich gegenseitig ergänzen. Es muss von Fall zu Fall abgewogen werden, welche Strategie den größten Sinn macht: eine Erweiterung oder eine Verknüpfung.

9.3 Zusammenfassung

Wie in den vorangegangenen Abschnitten gezeigt wurde, kann die Umsetzung eines programmbasierten Datenaustausches auch auf Basis einer anderen Programmiersprache (Syntax) wie beispielsweise der XML Query Language erfolgen. Auch andere Programmiersprachen wie Python oder BASIC-Dialekte könnten hier entsprechend adaptiert werden. Der programmbasierte Datenaustausch stellt allerdings nicht die einzige Möglichkeit dar, um Bauwerksdatenmodelle zu erweitern. Ansätze wie Linked-Data bieten die Möglichkeit, Bauwerksdatenmodelle mit spezifischen Informationen anzureichern bzw. unterschiedliche Datensätze miteinander zu verknüpfen. Die vorgestellten Ansätze weisen unterschiedliche Vor- und Nachteile auf und können zum Teil auch koexistieren.

Kapitel 10

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Vorschlag erarbeitet, Informationen über Bauwerke in Form von objektorientierten Programmen auszutauschen, um so Datenmodelle bezüglich ihrer Anforderungen flexibler zu gestalten. Dazu wurden die dedizierte Programmiersprache IFC-PL zur Umsetzung von Programmen entwickelt und eine entsprechende Laufzeitumgebung zusammen mit einem Integrationskonzept entwickelt. Dieses beschreibt, wie der IFC-PL-Ansatz in das bestehende Datenmodell Industry Foundation Classes (IFC) eingebettet werden kann. Darüber hinaus wurde gezeigt, wie der vorgestellte Ansatz abstrakte Konzepte beschreiben kann, die über die rein traditionelle Datenhaltung und -repräsentation von Bauwerksdaten hinausgehen. Dies wurde exemplarisch anhand der Abbildung von Richtlinien der RAS-L (Richtlinien für die Anlage von Straßen – Teil: Linienführung) veranschaulicht. Zudem wurden alternative Ansätze und Konzepte beleuchtet, die eine flexible Erweiterung von Bauwerkdatenmodellen erlauben.

10.1 Ergebnisse der Arbeit

Die wachsende Komplexität von Bauwerksmodellen und die zunehmende Digitalisierung von Bauprozessen stellt Softwareentwickler vor immer größere Herausforderungen. Neue Standards für Bauwerksdatenmodelle sollen kostengünstig, zeitnah und fehlerfrei umgesetzt werden, um so Open-BIM-Ansätze weiter voranzutreiben.

Mit dem in dieser Arbeit vorgestellten Konzept und der Realisierung dieses Konzeptes durch die IFC-PL wird eine Lösung aufgezeigt, die es ermöglicht, die Komplexität von Bauwerksdatenmodellen zu reduzieren. Dies wird dadurch erreicht, dass korrekte Implementierungsdetails hinter Schnittstellen verborgen, d. h. gekapselt werden. Durch Einführung dieser Schnittstellen wird ein höheres Abstraktionsniveau auf Seiten des Bauwerksdatenmodells erreicht. Die konkrete Ausgestaltung dieser Schnittstellen wird dabei den Softwareherstellern überlassen und

ist nicht mehr Teil des Bauwerksmodells selbst. Die Implementierungsdetails können mittels der IFC-PL- umgesetzt und in IFC-PL-Programmen ausgelagert werden. Durch die Schnittstellendefinition und IFC-PL-Laufzeitumgebung können konkrete Implementierungsdetails einer Schnittstelle zwischen unterschiedlichen Anwendungen ausgetauscht und genutzt werden, ohne dass diese vorher bekannt sein müssen. Eine Anwendung muss dabei nur die vereinbarte Schnittstelle auf Bauwerksmodellebene unterstützten. Dies hilft insgesamt, der wachsenden Komplexität von Bauwerksmodellen Herr zu werden und mit dieser besser umgehen zu können. Darüber hinaus bietet die Methode die Möglichkeit für Softwarehersteller, Schnittstellenimplementierungen an individuelle Anforderungen bei gleichzeitiger Gewährleistung des kontinuierlichen Datenflusses zwischen verschiedenen Anwendungen anzupassen. Im Rahmen der Arbeit und der prototypischen Umsetzung wurde gezeigt, dass ein solcher Lösungsansatz auf Basis der IFC-PL realisierbar ist.

Langwierige Standardisierungs- und Implementierungsprozesse, die kosten- und zeitaufwändig sind, lassen sich durch den IFC-PL-Ansatz zum Teil vermeiden. Abstrakte Schnittstellen, die nach den Best Practices des Softwaredesigns umgesetzt sind, lassen genug Spielraum für flexible Erweiterungen. Implementierungen von Schnittstellendefinitionen können unabhängig von einer Standardisierung entwickelt werden und lassen agile Anpassungen zu, die nicht an starre Standardisierungsprozesse gekoppelt sind. Die Details einer Implementierung selbst müssen dabei nicht standardisiert werden. Dies erlaubt im Rahmen der Schnittstellendefinition eine flexible Erweiterung von Bauwerksmodellen zur Laufzeit, ohne dass eine erneute Standardisierung oder Anpassung bestehender Softwareimplementierungen vorgenommen werden muss. Gleichzeit reduziert sich dadurch der Entwicklungsaufwand auf Seiten der Softwarehersteller, da nur die vereinbarten Schnittstellen auf Bauwerksmodellebene unterstützt und nicht alle möglichen Realisierungen und Ausprägungen dieser Schnittstellen umgesetzt werden müssen, diese aber trotzdem implizit durch den Ansatz unterstützt werden. Durch die prototypische Integration der IFC-PL-Laufzeitumgebung in unterschiedliche Softwarewerkzeuge wurde gezeigt, dass ein solcher Weg gangbar ist.

IFC-PL hilft, Softwarefehler, die z.B. durch Fehlinterpretation einer Dokumentation entstehen können, beim Datenaustausch zu vermeiden. Durch den Austausch von IFC-PL-Programmen kann die gleiche Implementierung einer Schnittstellenbeschreibung von verschiedenen Anwendungen verwendet werden. Dadurch werden die Daten, die hinter einer Schnittstellenimplementierung gekapselt sind, von allen Softwareanwendungen auf die gleiche Weise interpretiert und verarbeitet. Bei einem traditionellen Ansatz würde man auch die Attribute der Schnittstellenimplementierung mitstandardisieren. Diese Attribute müssten dann von jeder Anwendung selbst interpretiert und verarbeitet werden. Neben dem erhöhten Implementierungsaufwand können dabei auch Fehler bei der Umsetzung durch ein Softwarehaus gemacht werden. Eine nicht eindeutige Dokumentation der Attribute oder beliebige Programmierfehler bei den attributverarbeitenden Methoden können zu einem unerwünschtem Fehlverhalten führen.

10.2 Ausblick

Bauwerksdatenmodelle spielen im Kontext des Building Information Modelings eine zentrale Rolle. Im Bereich des Infrastrukturbaus fehlt es hier derzeit noch an umfassenden (Open-)BIM-Standards für den Tunnel-, Brücken-, Straßen- und Gleisbau. Zukünftige Forschungs- und Entwicklungsarbeiten können sich bei der Weiterentwicklung von Bauwerksdatenmodellen auf diese Domänen konzentrieren. Die Organisation buildingSMART scheint hierbei eine der treibendenden Kräfte zu sein, die es sich zum Ziel gesetzt hat, entsprechende Standards zu entwickeln und zu etablieren. Teilweise befinden sich diese auch schon in der Planung bzw. in der Realisierung, jedoch gibt es hierbei noch viele offene Fragestellungen (z. B.: Wie können möglichst viele Bauwerkstypen für eine bestimmte Domäne abgebildet werden, ohne die Komplexität eines Datenmodells drastisch zu erhöhen?).

Langwierige Standardisierungsvorhaben und zeitintensive Realisierungen von Standards durch Softwarehersteller sind typische Hindernisse, die Anwender daran hindern, die Potenziale und Ziele von Building Information Modeling ausschöpfen zu können. Im Rahmen dieser Arbeit wurden Möglichkeiten aufgezeigt, wie man diesen Problemen teilweise entgegenwirken kann. Dabei entsteht allerdings das Problem, dass mit diesen Programmen auch Intelligenz, die bisher in den Fachanwendungen verankert war, plötzlich in den Datenaustauschprozess miteingebunden wird. Es ist davon auszugehen, dass nicht jeder Softwarehersteller bereit ist, in dieser Form seinen Quellcode zu teilen. Feser & Rosenthal (2004) beschreiben diesen Umstand als einen nicht gewollten Know-How-Transfer. In weitergehenden Arbeiten könnten mögliche Auswege und Geschäftsmodelle (wie etwa Lizenzmodelle) untersucht werden, die mit dieser neuen Situation umgehen können. Beispielsweise könnte man IFC-PL-Programme nicht in Quelltextform, sondern als eine Art Zwischensprache (z. B. virtuelle Maschinensprache) weitergeben, um ein Reengineering durch Wettbewerber zu erschweren. Hierbei wäre es auch denkbar, Techniken aus dem Bereich der Code-Obfuscation (Dang *et al.*, 2014) anzuwenden. Zudem könnte man den Quelltext einfach durch ein geeignetes Verfahren verschlüsseln und so nur Käufern eines Lizenzschlüssels die Möglichkeit einräumen, bestimmte IFC-PL-Programme zu nutzen.

Auch im Bereich der Sprachkonzepte von IFC-PL können weitere Forschungsarbeiten unternommen werden. So könnte beispielsweise IFC-PL mit einer BIM-Anfragesprache wie etwa QL4BIM (Daum & Borrmann, 2015) erweitert werden, um beispielsweise ein weites Spektrum von Anwendungsfällen im Bereich des Code-Compliance-Checkings abzudecken.

Alternative Ansätze, welche beispielsweise die Standardisierung von offenen APIs für BIM-Services vorschlagen (Zeiss, 2014; van Berlo, 2014) oder Linked-Data-Ansätze (Pauwels *et al.*, 2015), bieten zukünftig sicherlich auch ein spannendes Forschungsumfeld. Die Fragestellung hierbei ist, wie die Datenhaltung für Bauwerke gestaltet werden muss, damit diese wirtschaftlich sinnvoll in BIM-Ansätze integriert werden kann und dabei das Datenmanagement über den Lebenszyklus eines Bauwerksmodells möglichst optimal unterstützt.

Anhang A

Literate Programming

Beim Literate Programming wird ein Programm in Fragmente aufgeteilt (Knuth, 1984). Beispielsweise kann ein Programm, das aus Modul- und Import-Anweisungen und einem Hauptprogramm besteht, wie folgt beschrieben werden:

⟨*Beispielprogramm*⟩≡
```
1    ⟨Modul und Import Anweisungen⟩
2    ⟨Hauptprogramm 206⟩
```

Der Name eines Fragments wird dabei in spitzen Klammern angegeben (z. B. ⟨Fragmentname⟩). Ein Fragment setzt sich aus Programmcode oder anderen Fragmenten zusammen. Dies wird mit dem Äquivalenzoperator ≡ beschrieben. Das angegebene Beispielprogramm besteht aus Modul- und Import-Anweisungen sowie einem Hauptprogramm. Die Zahl hinter einem Fragmentnamen gibt an, auf welcher Seitennummer das entsprechende Fragment definiert wird.

Das Modul- und Import-Anweisungsfragment wird im Folgenden definiert:

⟨*Modul- und Import-Anweisungen*⟩≡
```
1    module HelloWorld;
```

Obiges Fragment enthält eine Modul-Anweisung, die für das aktuelle Beispielprogramm die Zugehörigkeit zum HelloWorld-Modul festlegt. Ein bestehendes Fragment kann mit dem +≡ Operator erweitert werden:

⟨*Modul- und Import-Anweisungen*⟩+≡
```
1    import Core;
```

Das Fragment (⟨Modul- und Import-Anweisungen⟩) wurde mit einer Import-Anweisung erweitert.

Ziel des gezeigten Beispielprogramms soll die Ausgabe von „hello, world“ sein. Die Ausgabe dieses Textes wird im folgenden Fragment beschrieben:

⟨*Ausgabe*⟩≡
```
1    print("hello, world\n");
```

Im letzten Schritt wird das Hauptprogramm definiert.

⟨*Hauptprogramm*⟩≡
```
1     void main() {
2         ⟨Ausgabe 205⟩
3     }
```

Ein Literate-Programming-System kann aus der Eingabe, bestehend aus Fließtext und den Fragmenten, die in beliebiger Reihenfolge definiert werden können, wieder ein vollständiges, kompilierbares Programm zusammensetzen. Dieser Vorgang wird als tanglen (to tangle) bezeichnet. Ein Tangler könnte aus obiger Beschreibung folgenden Quelltext generieren:

```
1  // Beispielprogramm
2  // Modul und Import Anweisungen
3  module HelloWorld;
4
5  import Core;
6
7  // Hauptprogramm
8  void main() {
9      // Ausgabe
10     print("hello, world\n");
11 }
```

Das Ziel, das von Literate-Programming verfolgt wird, ist die Vereinigung von Dokumentation und Programmcode. Der Code soll nicht gut für den Computer, sondern gut für den menschlichen Programmierer lesbar bzw. verständlich sein. Dies macht insbesondere für längere IFC-PL-Programme im Rahmen dieser Arbeit Sinn und wird entsprechend im Kaptiel 8 und im Anhang B.4 angewendet.

Anhang B

IFC Programming Language

B.1 Schlüsselwörter

`Array`	`Bag`	`base`	`bool`	`break`
`case`	`cast`	`char`	`class`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`false`	`float`	`for`	`if`
`import`	`instanceof`	`int`	`interface`	`List`
`long`	`module`	`new`	`null`	`override`
`private`	`protected`	`public`	`return`	`Set`
`short`	`static`	`string`	`switch`	`this`
`throw`	`true`	`unknown`	`void`	`while`

B.2 Grammatik

Im Folgenden ist die Grammatik der IFC Programming Language (IFC-PL) in Backus-Naur-Form dargestellt. Die Produktionen sind alphabetisch sortiert. Das Startsymbol der Grammatik ist `program`.

⟨*access_modifier*⟩ ::= ⟨*ACCESS_MODIFIER*⟩

⟨*args_decl*⟩ ::= ⟨*/*intentionally_blank*/*⟩
  | ⟨*method_var_decl*⟩
  | ⟨*args_decl*⟩ ',' ⟨*method_var_decl*⟩

⟨*array_access_expr*⟩ ::= ⟨*ident*⟩ '[' ⟨*expr*⟩ ']' '=' ⟨*expr*⟩
  | ⟨*ident*⟩ '[' ⟨*expr*⟩ ']'

⟨*array_creation_expr*⟩ ::= 'new' ⟨*ident*⟩ '[' ⟨*expr*⟩ ']'

⟨*array_decl*⟩ ::= ⟨*ident*⟩ '[' ']' ⟨*ident*⟩
| ⟨*ident*⟩ '[' ']' ⟨*ident*⟩ '=' ⟨*array_creation_expr*⟩

⟨*block*⟩ ::= '{' ⟨*stmts*⟩ '}'
| '{' '}'

⟨*break_stmt*⟩ ::= 'break'

⟨*call_args*⟩ ::= ⟨/*blank*/⟩
| ⟨*expr*⟩
| ⟨*call_args*⟩ ',' ⟨*expr*⟩

⟨*cast_expr*⟩ ::= 'cast' '<' ⟨*ident*⟩ '>' '(' ⟨*expr*⟩ ')'

⟨*char_literal*⟩ ::= ⟨*CHRLIT*⟩

⟨*class_body*⟩ ::= ⟨*class_member*⟩
| ⟨*class_body*⟩ ⟨*class_member*⟩

⟨*class_decl*⟩ ::= 'class' ⟨*ident*⟩ '{' '}'
| 'class' ⟨*ident*⟩ '{' ⟨*class_body*⟩ '}'
| 'class' ⟨*ident*⟩ ':' ⟨*ident*⟩ '{' '}'
| 'class' ⟨*ident*⟩ ':' ⟨*ident*⟩ '{' ⟨*class_body*⟩ '}'

⟨*class_member*⟩ ::= ⟨*meth_decl*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ':' 'base' '(' ⟨*call_args*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ 'static' ⟨*var_decl*⟩ ';'
| ⟨*access_modifier*⟩ ⟨*var_decl*⟩ ';'
| ⟨*access_modifier*⟩ ⟨*array_decl*⟩ ';'

⟨*collection_parameter_decl*⟩ ::= ⟨*ident*⟩
| 'List' '<' ⟨*collection_parameter_decl*⟩ '>'
| 'Bag' '<' ⟨*collection_parameter_decl*⟩ '>'
| 'Set' '<' ⟨*collection_parameter_decl*⟩ '>'
| 'Array' '<' ⟨*collection_parameter_decl*⟩ '>'

⟨*continue_stmt*⟩ ::= 'continue'

⟨*do_while_stmt*⟩ ::= 'do' '{' '}' 'while' '(' ⟨*expr*⟩ ')'
| 'do' '{' ⟨*stmts*⟩ '}' 'while' '(' ⟨*expr*⟩ ')'

⟨*enum_body*⟩ ::= ⟨*ident*⟩
| ⟨*enum_body*⟩ ',' ⟨*ident*⟩

⟨*enum_decl*⟩ ::= 'enum' ⟨*ident*⟩ '{' ⟨*enum_body*⟩ '}'

⟨*expr*⟩ ::= 'new' ⟨*expr*⟩
| 'null'
| 'this'
| ⟨*expr*⟩ '.' ⟨*expr*⟩

| ⟨*expr*⟩ '+=' ⟨*expr*⟩
| ⟨*expr*⟩ '*=' ⟨*expr*⟩
| ⟨*ident*⟩ '(' ⟨*call_args*⟩ ')'
| ⟨*ident*⟩
| ⟨*numeric_literal*⟩
| ⟨*string_literal*⟩
| ⟨*char_literal*⟩
| ⟨*expr*⟩ '
| ⟨*expr*⟩ '=' ⟨*expr*⟩
| ⟨*expr*⟩ '*' ⟨*expr*⟩
| ⟨*expr*⟩ '/' ⟨*expr*⟩
| ⟨*expr*⟩ '+' ⟨*expr*⟩
| ⟨*expr*⟩ '-' ⟨*expr*⟩
| ⟨*expr*⟩ '==' ⟨*expr*⟩
| ⟨*expr*⟩ '!=' ⟨*expr*⟩
| ⟨*expr*⟩ '>=' ⟨*expr*⟩
| ⟨*expr*⟩ '<' ⟨*expr*⟩
| ⟨*expr*⟩ '>' ⟨*expr*⟩
| '(' ⟨*expr*⟩ ')'
| ⟨*ternary_conditional_exp*⟩
| ⟨*unary_expr*⟩
| ⟨*postfix_expr*⟩
| ⟨*prefix_expr*⟩
| ⟨*array_access_expr*⟩
| ⟨*cast_expr*⟩

⟨*for_init_stmt*⟩ ::= ⟨*var_decl*⟩
| ⟨*array_decl*⟩
| ⟨*expr*⟩

⟨*for_iterator*⟩ ::= ⟨*postfix_expr*⟩

⟨*for_stmt*⟩ ::= 'for' '(' ⟨*for_init_stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*for_iterator*⟩ ')' ⟨*stmt*⟩
| 'for' '(' ⟨*for_init_stmt*⟩ ';' ⟨*expr*⟩ ';' ⟨*for_iterator*⟩ ')' '{' '}'

⟨*func_decl*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*ident*⟩ ⟨*ident*⟩ '+' '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩

⟨*ident*⟩ ::= ⟨*TIDENIFIER*⟩

⟨*if_stmt*⟩ ::= 'if' '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩ 'else' ⟨*stmt*⟩
| 'if' '(' ⟨*expr*⟩ ')' ⟨*stmt*⟩

⟨*import_stmt*⟩ ::= 'import' ⟨*ident*⟩
| 'import' ⟨*ident*⟩ '.' ⟨*ident*⟩

⟨*instanceof_expr*⟩ ::= ⟨*expr*⟩ 'instanceof' ⟨*ident*⟩

⟨*interface_body*⟩ ::= ⟨*interface_method*⟩
| ⟨*interface_body*⟩ ⟨*interface_method*⟩

⟨*interface_decl*⟩ ::= ⟨*INTERFACE*⟩ ⟨*ident*⟩ '{' '}'
| ⟨*INTERFACE*⟩ ⟨*ident*⟩ '{' ⟨*interface_body*⟩ '}'

⟨*interface_method*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ';'
| ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' 'const' ';'

⟨*meth_decl*⟩ ::= ⟨*access_modifier*⟩ 'virtual' ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*REF*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*REF*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '[' ']' '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '+' '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '-' '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '*' '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ 'static' ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' 'override' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' 'const' 'override' ⟨*block*⟩
| ⟨*access_modifier*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '(' ⟨*args_decl*⟩ ')' 'const' ⟨*block*⟩

⟨*method_var_decl*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩
| 'const' ⟨*ident*⟩ ⟨*ident*⟩
| ⟨*REF*⟩ ⟨*ident*⟩ ⟨*ident*⟩
| ⟨*ident*⟩ '[' ']' ⟨*ident*⟩

⟨*module_stmt*⟩ ::= 'module' ⟨*ident*⟩

⟨*numeric_literal*⟩ ::= ⟨*INTEGER_LITERAL*⟩
| ⟨*FLOAT_LITERAL*⟩
| ⟨*DOUBLE_LITERAL*⟩

⟨*postfix_expr*⟩ ::= ⟨*ident*⟩ '++'
| ⟨*ident*⟩ '–'

⟨*prefix_expr*⟩ ::= '++' ⟨*ident*⟩

⟨*program*⟩ ::= ⟨*stmts*⟩

⟨*stmt*⟩ ::= ⟨*array_decl*⟩ ';'
| 'return' ⟨*expr*⟩ ';'
| 'return' ';'
| ⟨*block*⟩
| ⟨*break_stmt*⟩ ';'
| ⟨*class_decl*⟩
| ⟨*interface_decl*⟩
| ⟨*continue_stmt*⟩ ';'
| ⟨*enum_decl*⟩
| ⟨*expr*⟩ ';'
| ⟨*for_stmt*⟩
| ⟨*func_decl*⟩
| ⟨*if_stmt*⟩
| ⟨*import_stmt*⟩ ';'
| ⟨*var_decl*⟩ ';'
| ⟨*module_stmt*⟩ ';'
| ⟨*while_stmt*⟩
| ⟨*do_while_stmt*⟩ ';'
| ⟨*switch_stmt*⟩

⟨*stmts*⟩ ::= ⟨*stmt*⟩
  | ⟨*stmts*⟩ ⟨*stmt*⟩

⟨*string_literal*⟩ ::= ⟨*TSTRING*⟩

⟨*switch_label*⟩ ::= 'case' ⟨*expr*⟩ ':'
  | 'default' ':'

⟨*switch_section*⟩ ::= ⟨*switch_label*⟩ ⟨*stmts*⟩

⟨*switch_sections*⟩ ::= ⟨*switch_section*⟩
  | ⟨*switch_sections*⟩ ⟨*switch_section*⟩

⟨*switch_stmt*⟩ ::= 'switch' '(' ⟨*expr*⟩ ')' '{' '}'
  | 'switch' '(' ⟨*expr*⟩ ')' '{' ⟨*switch_sections*⟩ '}'

⟨*ternary_conditional_exp*⟩ ::= ⟨*expr*⟩ '?' ⟨*expr*⟩ ':' ⟨*expr*⟩

⟨*throw_stmt*⟩ ::= 'throw' 'new' ⟨*expr*⟩

⟨*unary_expr*⟩ ::= '-' ⟨*expr*⟩
  | '!' ⟨*expr*⟩

⟨*var_decl*⟩ ::= ⟨*ident*⟩ ⟨*ident*⟩
  | ⟨*ident*⟩ ⟨*ident*⟩ '=' ⟨*expr*⟩
  | ⟨*REF*⟩ ⟨*ident*⟩ ⟨*ident*⟩ '=' ⟨*expr*⟩
  | 'const' ⟨*ident*⟩ ⟨*ident*⟩ '=' ⟨*expr*⟩
  | 'List' '<' ⟨*collection_parameter_decl*⟩ '>' ⟨*ident*⟩ '=' 'new' 'List' '<' ⟨*collection_parameter_decl*⟩ '>' '(' ')'
  | 'Bag' '<' ⟨*collection_parameter_decl*⟩ '>' ⟨*ident*⟩ '=' 'new' 'Bag' '<' ⟨*collection_parameter_decl*⟩ '>' '(' ')'
  | 'Set' '<' ⟨*collection_parameter_decl*⟩ '>' ⟨*ident*⟩ '=' 'new' 'Set' '<' ⟨*collection_parameter_decl*⟩ '>' '(' ')'
  | 'Array' '<' ⟨*collection_parameter_decl*⟩ '>' ⟨*ident*⟩ '=' 'new' 'Array' '<' ⟨*collection_parameter_decl*⟩ '>' '(' ')'

⟨*while_stmt*⟩ ::= 'while' '(' ⟨*expr*⟩ ')' ⟨*stmts*⟩
  | 'while' '(' ⟨*expr*⟩ ')' '{' '}'

Die Symbole `TIDENIFIER`, `DOUBLE_LITERAL`, usw. können durch reguläre Ausdrücke beschrieben werden. Die Syntax der regulären Ausdrücke folgt hier der Konvention, die im Lexer-Framework flex üblich ist (siehe hierzu auch (Levine, 2009)).

```
ACCESS_MODIFIER ::= "public"|"private"|"protected"
CHRLIT ::= '([^'\\\n]|\\.)'{}
DOUBLE_LITERAL ::= [0-9]+\.[0-9]*
FLOAT_LITERAL ::= [0-9]+\.[0-9]*f
INTEGER_LITERAL ::= [0-9]+
TIDENIFIER ::= [a-zA-Z_][a-zA-Z0-9_]*
TSTRING ::= \"(\\.|[^"])*\"
```

Weiterhin ist folgende Operatorpräzedenz in Bison-Notation (Levine, 2009) definiert:

```
1 %left EQ /* == */
2 %left '='
3 %left '%'
4 %left '+' '-'
5 %left '*' '/'
6 %left INSTANCEOF
7 %left '.'
```

Dabei nimmmt die Operatorpräzedenz vom Vergleichsoperator (==) zum Punktoperator (.) zu.

B.3 Beispiele

B.3.1 Klothoide als Übergangskurve

B.3.1.1 Herleitung der x- und y-Koordinate

Die Krümmung K einer Klothoide nimmt linear mit der Länge der Klothoide L zu. Es gilt:

$$K = c \cdot L$$

Der Parameter c ist eine Konstante, die festlegt, wie schnell der Anstieg der Krümmung erfolgt. Typischerweise wird im Kontext der Trassierungsplanung der Klothoidenparameter A verwendet. Für diesen gilt:

$$\frac{1}{c} = A^2$$

Abbildung B.1 zeigt die Einheitsklothoide, bei der Klothoidenparameter A dem Wert 1 entspricht.

Aus diesen Randbedingungen lässt sich die folgende parametrische Beschreibung herleiten (Flurl, 2016):

$$x = \int_0^L \cos \frac{L^2}{2 \cdot A^2} \cdot dL$$

$$y = \int_0^L \sin \frac{L^2}{2 \cdot A^2} \cdot dL$$

Dabei handelt es sich um Fresnel'sche Integrale. Diese lassen sich mittels Reihenentwicklung oder durch numerische Integration lösen.

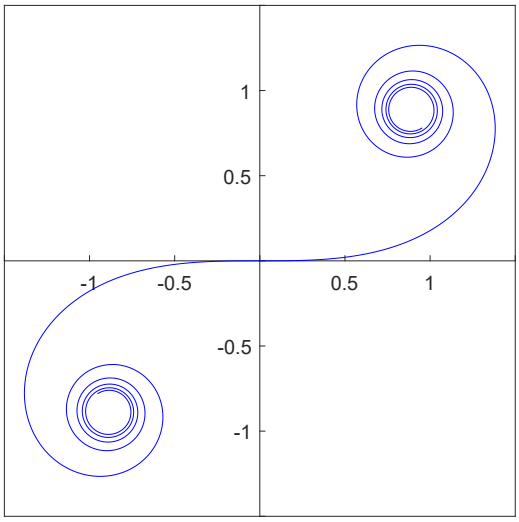


Abbildung B.1: Darstellung der Einheitsklothoide mit dem Parameter $A = 1$. Die Krümmung nimmt linear mit der Länge der Klothoide zu.

Die Reihenentwicklungen der Sinus- und Kosinusfunktion lauten:

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - ...$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} + ...$$

Durch Einsetzen der Reihenentwicklung erhält man für den x-Wert:

$$x = \int_0^L \cos \frac{L^2}{2 \cdot A^2} \cdot dL$$

$$x = \int_0^L (1 - \frac{L^4}{2! \cdot 4 \cdot A^4} + \frac{L^8}{4! \cdot 16 \cdot A^8} - \frac{L^{12}}{6! \cdot 64 \cdot A^{12}} + \frac{L^8}{8! \cdot 256 \cdot A^{16}} - +...) \cdot dL$$

$$x = L - \frac{L^5}{2 \cdot 4 \cdot 5 \cdot A^4} + \frac{L^9}{24 \cdot 16 \cdot 9 \cdot A^8} - \frac{L^{13}}{720 \cdot 64 \cdot 13 \cdot A^{12}} + \frac{L^{17}}{40320 \cdot 256 \cdot 17 \cdot A^{16}} - +...$$

$$x = L - \frac{L^5}{40 \cdot A^4} + \frac{L^9}{3456 \cdot A^8} - \frac{L^{13}}{599040 \cdot A^{12}} + \frac{L^{17}}{175472640 \cdot A^{16}} - +...$$

Für den y-Wert ergibt sich:

$$y = \int_0^L \sin \frac{L^2}{2 \cdot A^2} \cdot dL$$

$$y = \int_0^L (\frac{L^2}{1! \cdot 2 \cdot A^2} - \frac{L^6}{3! \cdot 8 \cdot A^6} + \frac{L^{10}}{5! \cdot 32 \cdot A^{10}} - \frac{L^{14}}{7! \cdot 128 \cdot A^{14}} + \frac{L^{18}}{9! \cdot 512 \cdot A^{14}} - +...) \cdot dL$$

$$y = \frac{L^3}{1 \cdot 2 \cdot 3 \cdot A^2} - \frac{L^7}{6 \cdot 8 \cdot 7 \cdot A^6} + \frac{L^{11}}{120 \cdot 32 \cdot 11 \cdot A^{10}} - \frac{L^{15}}{5040 \cdot 128 \cdot 15 \cdot A^{14}} + \frac{L^{19}}{362880 \cdot 512 \cdot 19 \cdot A^{14}} - +...$$

B.3.1.2 Implementierung

Beim Datenaustausch von Übergangskurven mittels des IFC-PL-Ansatzes (beschrieben in Sektion 8.1) werden konkret zwei Dateien ausgetauscht: eine Datei mit dem Namen Clothoid.ifcpl, die die Implementierung der Klasse Clothoid enthält, und eine STEP-P21-Datei, die entsprechend das IfcPropertySet enthält, das die Parameter für den Konstruktor der Klothoiden-Klasse bereithält.

Die Datei Clothoid.ifcpl umfasst folgenden Inhalt:

```
1  module Clothoid;
2
3  import IFC4X1.exp;
4  import Math;
5  import ArbitraryTransitionCurve;
6  import Core;
7
8  class Vector2DImpl : IVector2D {
9      public Vector2DImpl(const double x, const double y) {
10         this.x = x;
11         this.y = y;
12     }
13
14     public double X() const {
15         return x;
16     }
17     public double Y() const {
18         return y;
19     }
20
21     private double x;
22     private double y;
23 }
24
25 class Clothoid : IArbitraryTransitionCurve {
26     public Clothoid(IfcPropertySet ps) {
27         for(int i = 0; i < ps.HasProperties.count(); ++i) {
28             if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
29                 IfcPropertySingleValue s = cast<IfcPropertySingleValue>(ps.HasProperties[i]);
30                 IfcIdentifier s_id = s.Name;
31
32                 readReal(s, "StartDirection", startDirection_);
33                 readReal(s, "SegmentLength", length_);
34                 readReal(s, "startCurvature", startCurvature_);
35                 readReal(s, "clothoidConstant", clothoidConstant_);
36                 readBool(s, "counterClockwise", counterClockwise_);
37                 readBool(s, "entry", entry_);
38             }
39
40             if(ps.HasProperties[i] instanceof IfcPropertyListValue) {
41                 IfcPropertyListValue s = cast<IfcPropertyListValue>(ps.HasProperties[i]);
42                 IfcIdentifier s_id = s.Name;
43
44                 if(s_id.getValue() == "StartPoint") {
45                     IfcLengthMeasure x = cast<IfcLengthMeasure>(s.ListValues[0]);
46                     IfcLengthMeasure y = cast<IfcLengthMeasure>(s.ListValues[1]);
47
48                     startPosition_ = new Vector2d(x.getValue(), y.getValue());
49                 }
50             }
```

```
51          }
52
53          startL_ = clothoidConstant_ * clothoidConstant_ * startCurvature_;
54
55          if (entry_)
56              endL_ = startL_ + length_;
57          else
58              endL_ = max(startL_ - length_, 0.0);
59      }
60
61      private void readReal(IfcPropertySingleValue ifcPSV, string name,
62                            ref double outValue) {
63          IfcIdentifier s_id = ifcPSV.Name;
64
65          if(s_id.getValue() == name) {
66              IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
67              outValue = ifcReal.getValue();
68          }
69      }
70
71      private void readBool(IfcPropertySingleValue ifcPSV, string name,
72                            ref bool outValue) {
73          IfcIdentifier s_id = ifcPSV.Name;
74
75          if(s_id.getValue() == name) {
76              IfcBoolean ifcBoolean = cast<IfcBoolean>(s.NominalValue);
77              outValue = ifcBoolean.getValue();
78          }
79      }
80
81      public double getLength() const override {
82          return abs(endL_ - startL_);
83      }
84
85      public IVector2D getPostion(const double lerpParameter) const override {
86          Vector2d localPosition = new Vector2d();
87          Vector2d localOffset = new Vector2d();
88
89          double L = startL_ + (endL_ - startL_) * lerpParameter;
90
91          localPosition = computeLocalPosition(L);
92
93          double angle;
94
95          Vector2d position;
96          angle = computeTau(startL_, clothoidConstant_);
97          localOffset = localPosition - computeLocalPosition(startL_);
98
99          if (!isEntry()) {
100             angle *= -1;
101             localOffset.x() *= -1;
102         }
103
104         if (!counterClockwise_) {
105             angle *= -1;
106         }
107
108         position = startPosition_ + createRotationMatrix(startDirection_ - angle) * localOffset;
109
110         return new Vector2DImpl(position.x(), position.y());
111     }
112
113     public double getCurvature(const double lerpParamter) const override {
114         double L = startL_ + (endL_ - startL_) * lerpParameter;
115         return L / (A * A);
116     }
117
118     private double computeX(const double L, const double A) const {
119         double x = L;
120         const int iterations = 5;
121
122         for (int i = 1; i < iterations+1; i++) {
123             double sign = i % 2 == 0 ? 1 : -1;
124
125             double L_exponent = 5+(i-1)*4;
126             double A_exponent = i*4;
127             double factor = factorial(2*i) * pow(2.0, 2*i) * (5+(i-1)*4);
128             factor = factor + 3.0;
129             double debug = pow(A, A_exponent);
130             x += sign * pow(L, L_exponent) / (factor * debug);
131         }
132
133         return x;
134     }
135
136     private double computeY(const double L, const double A) const {
137         double y = 0;
138         const int iterations = 5;
139
140         for (int i = 0; i < iterations; i++)
```

```
141         {
142             double sign = i % 2 == 0 ? 1 : -1;
143 
144             double L_exponent = 3+i*4;
145             double A_exponent = 2+i*4;
146             double factor = factorial(2*i+1) *
147                 pow(2.0, i*2) * 2 *
148                 (A_exponent + 1);
149 
150             y += sign * pow(L, L_exponent) / (factor * pow(A, A_exponent));
151         }
152 
153         return y;
154     }
155 
156     private bool isEntry() const {
157         return entry_;
158     }
159 
160     private Vector2d computeLocalPosition(const double L) const {
161         Vector2d localPosition = new Vector2d();
162 
163         localPosition.x() = computeX(L, clothoidConstant_);
164         localPosition.y() = computeY(L, clothoidConstant_);
165 
166         return localPosition;
167     }
168 
169     private double computeTau(const double L, const double A) const {
170         return L*L / (2 * A*A);
171     }
172 
173     public IfcPropertySet getParamters() {
174         IfcPropertySingleValue StartCurvature = new IfcPropertySingleValue();
175         StartCurvature.Name = new IfcIdentifier("startCurvature");
176         StartCurvature.NominalValue = new IfcReal(startCurvature_);
177 
178         IfcPropertySingleValue CounterClockwise = new IfcPropertySingleValue();
179         CounterClockwise.Name = new IfcIdentifier("counterClockwise");
180         CounterClockwise.NominalValue = new IfcBoolean(counterClockwise_);
181 
182         IfcPropertySingleValue ClothoidConstant = new IfcPropertySingleValue();
183         ClothoidConstant.Name = new IfcIdentifier("clothoidConstant");
184         ClothoidConstant.NominalValue = new IfcReal(clothoidConstant_);
185 
186         IfcPropertySingleValue Entry = new IfcPropertySingleValue();
187         Entry.Name = new IfcIdentifier("entry");
188         Entry.NominalValue = new IfcBoolean(entry_);
189 
190         IfcPropertySet ps = new IfcPropertySet();
191         ps.Name = new IfcLabel("Parameters");
192         ps.HasProperties.add(StartCurvature);
193         ps.HasProperties.add(ClothoidConstant);
194         ps.HasProperties.add(CounterClockwise);
195         ps.HasProperties.add(Entry);
196 
197         return ps;
198     }
199 
200     // injected data
201     private Vector2d startPosition_;
202     private double startDirection_;
203     private double startCurvature_;
204     private bool counterClockwise_;
205     private double clothoidConstant_;
206     private bool entry_;
207     private double length_;
208 
209     // misc
210     private double startL_;
211     private double endL_;
212 }
```

B.3.2 Ein Doppel-T-Profil als parametrische Profildefinition

```
1  module IShapeProfile;
2 
3  import ParametricProfileDescription;
4  import IFC4X1.exp;
5 
6  class IShapeProfile : IParametricProfileDescription {
7      // IShapeProfile constructor
8      IShapeProfile(IfcPropertySet ps) {
9          for(int i = 0; i < ps.HasProperties.count(); ++i) {
10             if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
11                 IfcPropertySingleValue s = cast<IfcPropertySingleValue>(ps.HasProperties[i]);
```

```
12                  IfcIdentifier s_id = s.Name;
13
14                  readReal(s, "OverallWidth", OverallWidth_);
15                  readReal(s, "OverallDepth", OverallDepth_);
16                  readReal(s, "WebThickness", WebThickness_);
17                  readReal(s, "FlangeThickness", FlangeThickness_);
18              }
19          }
20      }
21
22      private void readReal(IfcPropertySingleValue ifcPSV, string name,
23                            ref double outValue) {
24          IfcIdentifier s_id = ifcPSV.Name;
25
26          if(s_id.getValue() == name) {
27              IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
28              outValue = ifcReal.getValue();
29          }
30      }
31
32      // IShapeProfile getParameters
33      public IfcPropertySet getParameters() {
34          IfcPropertySingleValue OverallWidth = new IfcPropertySingleValue();
35          OverallWidth.Name = new IfcIdentifier("OverallWidth");
36          OverallWidth.NominalValue = new IfcReal(OverallWidth_);
37
38          IfcPropertySingleValue OverallDepth = new IfcPropertySingleValue();
39          OverallDepth.Name = new IfcIdentifier("OverallDepth");
40          OverallDepth.NominalValue = new IfcReal(OverallDepth_);
41
42          IfcPropertySingleValue WebThickness = new IfcPropertySingleValue();
43          WebThickness.Name = new IfcIdentifier("WebThickness");
44          WebThickness.NominalValue = new IfcReal(WebThickness_);
45
46          IfcPropertySingleValue FlangeThickness = new IfcPropertySingleValue();
47          FlangeThickness.Name = new IfcIdentifier("FlangeThickness");
48          FlangeThickness.NominalValue = new IfcReal(FlangeThickness_);
49
50          IfcPropertySet ps = new IfcPropertySet();
51          ps.Name = new IfcLabel("Parameters");
52          ps.HasProperties.add(OverallWidth);
53          ps.HasProperties.add(OverallDepth);
54          ps.HasProperties.add(WebThickness);
55          ps.HasProperties.add(FlangeThickness);
56
57          return ps;
58      }
59
60      // IShapeProfile getBasicProfileDescription
61      public IfcProfileDef getBasicProfileDescription() {
62          IfcPolyline polyline = new IfcPolyline();
63
64          // IShapeProfile compute points
65          IfcCartesianPoint[] points = new IfcCartesianPoint[12];
66          points[0] = createPoint(Left,Top); // A
67          points[1] = createPoint(Right,Top); // B
68          points[2] = createPoint(Right,Top−FlangeThickness_); // C
69          points[3] = createPoint(WebThickness_∗0.5,Top−FlangeThickness_); // D
70          points[4] = createPoint(WebThickness_∗0.5,Bottom+FlangeThickness_); // E
71          points[5] = createPoint(Right,Bottom+FlangeThickness_); // F
72          points[6] = createPoint(Right,Bottom); // G
73          points[7] = createPoint(Left,Bottom); // H
74          points[8] = createPoint(Left,Bottom+FlangeThickness_); // I
75          points[9] = createPoint(−WebThickness_∗0.5,Bottom+FlangeThickness_); // J
76          points[10] = createPoint(−WebThickness_∗0.5,Top−FlangeThickness_); // K
77          points[11] = createPoint(Left,Top−FlangeThickness_); // L
78
79          for(int i = 0; i < 12; i++) {
80              polyline.Points.add(points[i]);
81          }
82
83          // close profile
84          polyline.Points.add(points[0]); // add point A
85
86          IfcArbitraryClosedProfileDef profile = new IfcArbitraryClosedProfileDef();
87
88          profile.ProfileType = new IfcProfileTypeEnum(IfcProfileTypeEnum.AREA);
89          profile.OuterCurve = polyline;
90
91          return profile;
92      }
93
94      // IShapeProfile createPoint
95      private IfcCartesianPoint createPoint(const double x, const double y) {
96          IfcCartesianPoint point = new IfcCartesianPoint();
97
98          point.Coordinates.add(new IfcLengthMeasure(x));
99          point.Coordinates.add(new IfcLengthMeasure(y));
100
101         return point;
```

```
102     }
103
104     // IShapeProfile member variables
105     private double OverallWidth_;
106     private double OverallDepth_;
107     private double WebThickness_;
108     private double FlangeThickness_;
109 }
```

B.3.3 Rechtwinkliges Kastenwiderlager

```
1   module Abutment;
2
3   import Core;
4   import Math;
5   import IAbutment;
6   import IFC4X1.exp;
7
8   class Abutment : IAbutment {
9       // Abutment class constructor
10      public Abutment(IfcPropertySet ps) {
11          for(int i = 0; i < ps.HasProperties.count(); ++i) {
12              if (ps.HasProperties[i] instanceof IfcPropertySingleValue) {
13                  IfcPropertySingleValue s =
14                      cast<IfcPropertySingleValue>(ps.HasProperties[i]);
15                  IfcIdentifier s_id = s.Name;
16
17                  readReal(s, "width", width_);
18                  readReal(s, "length", length_);
19                  readReal(s, "innerWingWidth", innerWingWidth_);
20                  readReal(s, "innerEndWallWidth", innerEndWallWidth_);
21                  readReal(s, "height", height_);
22                  readReal(s, "wingWidthA", wingWidthA_);
23                  readReal(s, "wingWidthB", wingWidthB_);
24                  readReal(s, "wingHeightA", wingHeightA_);
25                  readReal(s, "wingHeightB", wingHeightB_);
26              }
27          }
28      }
29
30      // Abutment class methods
31      public IfcSolidModel getSolidModel() {
32          IfcBooleanResult br = new IfcBooleanResult();
33          br.Operator = new IfcBooleanOperator(IfcBooleanOperator.DIFFERENCE);
34          br.FirstOperand = createBaseMesh();
35          br.SecondOperand = createAuxiliaryMesh();
36
37          IfcCsgSolid csgSolid = new IfcCsgSolid();
38          csgSolid.TreeRootExpression = br;
39
40          return csgSolid;
41      }
42
43      private IfcExtrudedAreaSolid createBaseMesh() {
44          IfcExtrudedAreaSolid baseMesh = new IfcExtrudedAreaSolid();
45          baseMesh.SweptArea = createGroundArea();
46          baseMesh.ExtrudedDirection = createDirecton(0, 0, 1);
47          baseMesh.Depth = new IfcPositiveLengthMeasure(scale_ * height_);
48          return baseMesh;
49      }
50
51      private IfcArbitraryClosedProfileDef createGroundArea() {
52          IfcPolyline polyline = new IfcPolyline();
53
54          Vector2d[] points = new Vector2d[9];
55
56          /*A*/ points[0] = new Vector2d(0, -width_ / 2.0);
57          /*B*/ points[1] = new Vector2d(0, width_ / 2.0);
58          /*C*/ points[2] = new Vector2d(length_, width_ / 2.0);
59          /*D*/ points[3] = new Vector2d(length_, width_ / 2.0 - innerWingWidth_);
60          /*E*/ points[4] = new Vector2d(innerEndWallWidth_,
61                                         width_ / 2.0 - innerWingWidth_);
62          /*F*/ points[5] = new Vector2d(innerEndWallWidth_,
63                                         -width_ / 2.0 + innerWingWidth_);
64          /*G*/ points[6] = new Vector2d(length_, -width_ / 2.0 + innerWingWidth_);
65          /*H*/ points[7] = new Vector2d(length_, -width_ / 2.0);
66
67          // The OuterCurve has to be a closed curve.
68          points[8] = new Vector2d(0, -width_ / 2.0); // A
69
70          for(int i = 0; i < 9; i++) {
71              polyline.Points.add(createPoint2D(scale_ * points[i]));
72          }
73
74          IfcArbitraryClosedProfileDef closedProfile =
75              new IfcArbitraryClosedProfileDef();
76          closedProfile.OuterCurve = polyline;
77          closedProfile.ProfileType =
```

```
78              new IfcProfileTypeEnum(IfcProfileTypeEnum.AREA);
79
80          return closedProfile;
81      }
82
83      private IfcDirection createDirecton(double x, double y, double z) {
84          IfcDirection dir = new IfcDirection();
85          dir.DirectionRatios.add(new IfcReal(x));
86          dir.DirectionRatios.add(new IfcReal(y));
87          dir.DirectionRatios.add(new IfcReal(z));
88          return dir;
89      }
90
91      private IfcExtrudedAreaSolid createAuxiliaryMesh() {
92          IfcAxis2Placement3D placement = new IfcAxis2Placement3D();
93          placement.Location = createPoint3D(new Vector3d(0, scale_ * width_ / 2.0, 0));
94          placement.Axis = createDirecton(0.0, -1.0, 0.0);
95
96          IfcExtrudedAreaSolid baseMesh = new IfcExtrudedAreaSolid();
97          baseMesh.SweptArea = createAuxArea();
98          baseMesh.ExtrudedDirection = createDirecton(0, 0, 1);
99          baseMesh.Depth = new IfcPositiveLengthMeasure(scale_ * height_);
100         baseMesh.Position = placement;
101         return baseMesh;
102     }
103
104     private IfcArbitraryClosedProfileDef createAuxArea() {
105         IfcPolyline polyline = new IfcPolyline();
106
107         Vector2d[] points = new Vector2d[6];
108
109         /* A' */ points[0] = new Vector2d(length_ - wingWidthA_, 0);
110         /* B' */ points[1] = new Vector2d(length_ - wingWidthA_, wingHeightA_);
111         /* C' */ points[2] = new Vector2d(length_ - wingWidthB_, wingHeightB_);
112         /* D' */ points[3] = new Vector2d(length_, wingHeightB_);
113         /* E' */ points[4] = new Vector2d(length_, 0.0);
114         // The OuterCurve has to be a closed curve.
115         /* A' */ points[5] = new Vector2d(length_ - wingWidthA_, 0);
116
117         for(int i = 0; i < 6; i++) {
118             polyline.Points.add(createPoint2D(scale_ * points[i]));
119         }
120
121         IfcArbitraryClosedProfileDef closedProfile = new IfcArbitraryClosedProfileDef();
122         closedProfile.OuterCurve = polyline;
123         closedProfile.ProfileType = new IfcProfileTypeEnum(IfcProfileTypeEnum.AREA);
124
125         return closedProfile;
126     }
127
128     private IfcCartesianPoint createPoint3D(Vector3d v) {
129         IfcCartesianPoint point = new IfcCartesianPoint();
130         point.Coordinates.add(new IfcLengthMeasure(v.x()));
131         point.Coordinates.add(new IfcLengthMeasure(v.y()));
132         point.Coordinates.add(new IfcLengthMeasure(v.z()));
133         return point;
134     }
135
136     private IfcCartesianPoint createPoint2D(Vector2d v) {
137         IfcCartesianPoint point = new IfcCartesianPoint();
138         point.Coordinates.add(new IfcLengthMeasure(v.x()));
139         point.Coordinates.add(new IfcLengthMeasure(v.y()));
140         return point;
141     }
142
143     private void readReal(IfcPropertySingleValue ifcPSV, string name,
144                           ref double outValue) {
145         IfcIdentifier s_id = ifcPSV.Name;
146
147         if(s_id.getValue() == name) {
148             IfcReal ifcReal = cast<IfcReal>(ifcPSV.NominalValue);
149             outValue = ifcReal.getValue();
150         }
151     }
152
153     // Abutment class member variables
154     private double width_ = 5;
155     private double length_ = 4.5;
156     private double innerWingWidth_ = 1;
157     private double innerEndWallWidth_ = 1;
158     private double height_ = 5;
159     private double wingWidthA_ = 2.5;
160     private double wingWidthB_ = 1.0;
161     private double wingHeightA_ = 3.0;
162     private double wingHeightB_ = 4.0;
163
164     private double scale_ = 1000;
165 }
```

B.3.4 RALChecker

```
1  module RALChecker;
2
3  import Core;
4  import IFC4X1.exp;
5
6  class RALChecker {
7      public RALChecker(IfcAlignment a) {
8          alignment_ = a;
9      }
10
11     public void check() {
12         visit(alignment_);
13     }
14
15     private void visit(IfcCircularArcSegment2D arcSegment) {
16         bool reportError = false;
17
18         if(arcSegment.Radius.getValue() < 500) {
19             print("Rule (Radius >= 500) is violated!");
20             reportError = true;
21         }
22
23         if(arcSegment.SegmentLength.getValue() < 70) {
24             print("Rule (Length >= 70) is violated!");
25             reportError = true;
26         }
27
28         if(reportError) {
29             if(alignment_.Name != null) {
30                 print("Alignment name: " + alignment_.Name.getValue());
31             }
32
33             print("Station: " + currentStation_);
34         }
35     }
36
37     private void visit(IfcAlignment2DHorizontalSegment hs) {
38         IfcCurveSegment2D cs = hs.CurveGeometry;
39
40         if(cs instanceof IfcCircularArcSegment2D ) {
41             IfcCircularArcSegment2D arcSegment =
42                 cast<IfcCircularArcSegment2D>(cs);
43             visit(arcSegment);
44         }
45
46         currentStation_ += cs.SegmentLength.getValue();
47     }
48
49     private void visit(IfcAlignmentCurve ac) {
50         IfcAlignment2DHorizontal ha = ac.Horizontal;
51         currentStation_ = ha.StartDistAlong.getValue();
52         for(int i = 0; i < ha.Segments.count(); i++) {
53             visit(ha.Segments[i]);
54         }
55     }
56
57     private void visit(IfcAlignment a) {
58         if(a.Axis instanceof IfcAlignmentCurve) {
59             IfcAlignmentCurve ac = cast<IfcAlignmentCurve>(a.Axis);
60             visit(ac);
61         }
62         else {
63             print("Unkown alignment curve type.");
64         }
65     }
66
67     private double currentStation_ = 0;
68     private IfcAlignment alignment_;
69 }
70
71 void main(string[] args) {
72     if(args.count() != 1) {
73         print("Program called with invalid number of arguments");
74         return;
75     }
76
77     string filename = args[0];
78
79     if(!exists(filename)) {
80         print("File does not exist.");
81         return;
82     }
83
84     print("Try to load " + filename);
85     IfcStepReader reader = new IfcStepReader();
86     IFC4X1Model model = reader.read(filename);
87
88     for(int i = 0; i < model.getEntitiesCount(); i++) {
```

```
89          IFC4X1Entity e = model.getEntityByIndex(i);
90
91          if(e instanceof IfcAlignment) {
92              IfcAlignment a = cast<IfcAlignment>(e);
93
94              RALChecker checker = new RALChecker(a);
95              checker.check();
96          }
97      }
98  }
```

B.4 Beweis der Turing-Vollständigkeit

Die IFC-PL ist Turing-vollständig. In diesem Abschnitt soll diese Aussage bewiesen werden. Dazu wird gezeigt, wie jede beliebige Turingmaschine (Turing, 1937) mithilfe eines IFC-PL-Programms simuliert werden kann. Außerdem dient dieser Simulator als Beispiel für ein etwas umfangreicheres IFC-PL-Programm.

Eine Turingmaschine ist formal betrachtet ein 7-Tuple $(Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, bestehend aus:

- einer endlichen Zustandsmenge Q
- einem endlichen Eingabealphabet Σ
- einem endlichen Bandalphabet Γ mit $\Sigma \subset \Gamma$
- einer partiellen Überführungsfunktion $\delta\colon (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$
- einem Anfangszustand q_0 mit $q_0 \in Q$
- dem leeren Feld $\square$, wobei $\square \in \Gamma \setminus \Sigma$ gilt
- der Menge der akzeptierenden Endzustände F mit $F \subseteq Q$

Die Turingmaschine liest, beginnend in ihrer Startkonfiguration, das erste Zeichen vom Band und führt dann entsprechend ihrer Konfiguration und der partiell definierten Übergangsfunktion Aktionen durch. Diese umfassen das Überschreiben des aktuell gelesenen Bandsymbols und das Bewegen des Schreib-/Lesekopfs (der Schreib-/Lesekopf kann auch an der aktuell befindlichen Stelle verweilen). Grundlagen zu Turingmaschinen bzw. allgemeinen Konzepten der theoretischen Informatik finden sich in zahlreichen Publikationen behandelt und sollen an dieser Stelle nicht wiederholt werden. Beispielsweise werden Grundlagen zur Automatentheorie und insbesondere zur Berechenbarkeit in (Kozen, 1997), (Socher, 2008) oder (Hopcroft, 2007) beschrieben.

B.4.1 Turingmaschinensimulator

Da das Programm des Turingmaschinensimulators umfangreicher ist, wird dieses mithilfe von Literate Programming beschrieben. Nähere Informationen zu dieser Technik finden sich im Anhang A.

Der Turingmaschinensimulator ist in sechs Teilabschnitte gegliedert.

⟨*TuringMachineSimulator*⟩≡

```
1   ⟨Module and import statemnts⟩
2   ⟨Direction enumeration 222⟩
3   ⟨Transition class 224⟩
```

```
4     ⟨State class 222⟩
5     ⟨Turing machine class 224⟩
6     ⟨Main function 229⟩
```

Das Programm befindet sich innerhalb des Moduls TuringMachineSimulator und ist auf einfache Textausgaben (print-Funktion) des Moduls Core angewiesen.

⟨*Module and import statements*⟩≡

```
1 module TuringMachineSimulator;
2
3 import Core;
```

Eine Turingmaschine besitzt einen Lese-Schreibkopf, der nach jedem Schritt das Speicherband wahlweise nach links oder rechts bewegen kann. Diese Bewegung ist optional. Der Lese-Schreibkopf kann genauso an der aktuellen Position auf dem Band stehenbleiben und sich nicht bewegen. Die Bewegung des Speicherbands wird mithilfe des selbstdefinierten Aufzählungdatentyps eDirection beschrieben:

⟨*Direction enumeration*⟩≡

```
1 enum eDirection {
2     Left,
3     Right,
4     Hold
5 }
```

Der Wert Left des Aufzählungsdatentyps beschreibt eine Linksbewegung des Speicherbands. Entsprechend beschreibt der Wert Right eine Rechtsbewegung des Speicherbands und der Wert Hold ein Stehenbleiben an der aktuellen Position.

Von zentraler Bedeutung im IFC-PL-Turingmaschinensimulator ist die Klasse State.

⟨*State class*⟩≡

```
1 class State {
2     ⟨State class ctor 223⟩
3
4     ⟨State class methods 223⟩
5
6     ⟨State class members 222⟩
7 }
```

Verschiedene Zustände werden mithilfe eines eindeutigen Identifiers (id) vom Datentyp Integer unterschieden:

⟨*State class members*⟩≡

```
1 public int id;
```

Natürlich ist der Datentyp Integer in seinem Wertebereich beschränkt und kann daher keine Turingmaschinen abbilden, die mehr Zustände umfassen, als der Wertebereich diese Datentyps zulässt. Allerdings kann dieser Umstand trivial durch die Einführung eines mächtigeren Id-Types umgangen werden und wird ohne Beschränkung der Allgemeinheit in diesem Abschnitt entsprechend weitergeführt.

Zudem merkt sich jeder Zustand die mit ihm verknüpften Transitionen. Auch hier wurde wieder eine Vereinfachung getroffen. Die maximale Anzahl der Transitionen wurde auf fünf beschränkt und die verschiedenen Transitionen werden in einem Array verwaltet. Auch in diesem Fall bleibt dieser Beweis ohne Beschränkung der Allgemeinheit gültig, da die Anzahl der möglichen Transitionen trivial erweitert werden kann.

⟨*State class members*⟩+≡
```
1 public int transitionCount;
2 public const int MAX_TRANSITIONS = 5;
3 public Transition [] transitions = new Transition[MAX_TRANSITIONS];
```

Weiter besitzt die Zustandsklasse ein Attribut, das festlegt, ob der entsprechende Zustand ein Endzustand ist.

⟨*State class members*⟩+≡
```
1 public bool accept;
```

Daneben definiert die Klasse State ein statisches Attribut:

⟨*State class members*⟩+≡
```
1 private static int stateId = 0;
```

Intern wird die Variable stateId als globaler Zähler verwendet, der dafür sorgt, dass jeder Zustand eine eindeutige Kennung in Form eines Ganzzahlwertes erhält. Dieser Zähler wird im Konstruktor der Stateklasse inkrementiert. Außerdem nimmt dieser noch weitere Initialisierungen vor.

⟨*State class ctor*⟩≡
```
1 public State(const bool accept) {
2     this.id = stateId++;
3     this.accept = accept;
4     this.transitionCount = 0;
5 }
```

Die State-Klasse besitzt nur eine Methode, mit deren Hilfe Transitionen zum aktuellen Zustand hinzugefügt werden können. Die Anzahl der Transitionen ist zwar programmtechnisch auf eine feste Anzahl limitiert (MAX_TRANSITIONS), lässt sich aber trivial erweitern und stellt somit keine Einschränkung der simulierbaren Turingmaschinen dar.

⟨*State class methods*⟩≡
```
1 public void add(Transition ts) {
2     if(transitionCount == MAX_TRANSITIONS) {
3         throw new Exception("State already has the maximum amount of
            transitions.");
4     }
5
6     // add the transition
7     transitions[transitionCount] = ts;
8     transitionCount++;
9 }
```

Formal wird eine Turingmaschine u.a. mithilfe einer (partiellen) Übergangsfunktion beschrieben. Diese bildet eine endliche Zustandsmenge (ohne akzeptierende Endzustände) zusammen mit einem endlichen Bandalphabet auf ein Triple, bestehend aus einem Folgezustand (inklusive akzeptierende Endzustände), einem Element des Bandalphabets und einer Lese-Schreibkopfbewegung ab. Ein Element dieser Relation wird mithilfe der Klasse Transition beschrieben.

⟨*Transition class*⟩≡
```
1 class Transition {
2     ⟨Transition class ctor 224⟩
3
4     ⟨Transition class members 224⟩
5 }
```

Die Klasse Transition legt fest, welches Eingabesymbol (input) mit welchem Ausgabesymbol (write) überschrieben werden, in welche Richtung sich der Lese-Schreib-Kopf bewegen und welcher Folgezustand im Anschluss eingenommen werden soll. Auch hier sind wieder Vereinfachungen getroffen worden, die aber keine allgemeinen Beschränkungen des Beweises darstellen. Die Bandsymbole wurden z. B. auf den Wertebereich des Datentyps char beschränkt. Diese Beschränkung lässt sich aber auf einfache Weise aufheben.

⟨*Transition class members*⟩≡
```
1 public char input;
2 public char write;
3 public eDirection move;
4 public State next;
```

Der entsprechende Konstruktor initialisiert die Attribute.

⟨*Transition class ctor*⟩≡
```
1 public Transition(const char input, const char write, const eDirection
    move, const State next) {
2     this.input = input;
3     this.write = write;
4     this.move = move;
5     this.next = next;
6 }
```

Die zentrale Klasse des Simulators ist die Klasse TuringMachine.

⟨*Turing machine class*⟩≡
```
1 class TuringMachine {
2     ⟨TuringMachine class ctor 225⟩
3     ⟨TuringMachine class methods 225⟩
4     ⟨TuringMachine class members 224⟩
5 }
```

Die Klasse TuringMachine verwaltet intern den aktuellen Zustand und den initialen Zustand des Simulators.

⟨*TuringMachine class members*⟩≡
```
1 private State current;
```

```
2  private State initialState;
```

Bei der hier beschriebenen Turingmaschine handelt es sich um eine einseitig beschränkte Turingmaschine. Das Band ist links beschränkt und nur rechts beschreibbar. Einseitig beschränkte Turingmaschinen können durch Turingmaschinen mit nichtbeschränktem Band, d.h. mit links und rechts beschreibarem Band, simuliert werden (Illik, 2009). Darüber hinaus könnte der Quellcode des Simulators durch eine einfache Erweiterung auch linksseitig auf das Band schreiben. Jedoch wird hier darauf verzichtet, um diesen Beweis nicht in die Länge zu ziehen.

Das Band wird mittels des Datentyps string modelliert. Die damit einhergehende Limitierung des Bandalphabets auf gültige Zeichen, die vom Datentyp String unterstützt werden, stellt ebenfalls ein trivial zu lösendes Problem dar. Beispielsweise könnten weitere Zeichen durch eine bestimmte Kodierung implizit unterstützt werden. Die Variable tapeLength speichert die maximale Länge des Bandes der Turingmaschine. Der Speicherplatz des Bandes ist physikalisch begrenzt, genauso wie der endliche Speicher des Rechners, auf dem der Simulator läuft. Wir nehmen hier einfach an, dass der Rechner über unendlich viel Speicher verfügt und mit einem Höhersetzten der Bandlänge beliebig viel Speicher auf dem Band zur Verfügung steht. Ein weiteres wichtiges Attribut der Klasse stellt die Variable head dar. Diese merkt sich die aktuelle Position des Lese-Schreibkopfs auf dem Band.

⟨*TuringMachine class members*⟩+≡

```
1  private int tapeLength;
2  private string tape;
3  private int head;
```

Der Konstruktor initialisiert alle Variablen.

⟨*TuringMachine class ctor*⟩≡

```
1  public TuringMachine() {
2      tapeLength = 0;
3      tape = "";
4      current = null;
5      head = 0;
6  }
```

Die Klasse bietet darüberhinaus verschiedene Methoden an. Für Debug- und Fehlerausgaben wurden die beiden Methoden debug und die eingeführt.

⟨*TuringMachine class methods*⟩≡

```
1  private void debug(string str) {
2      print(str);
3  }
4
5  private void die(string str) {
6      throw new Exception(str);
7  }
```

Mit der Methode setInitialState kann der initiale Startzustand der Turingmaschine festgelegt werden.

⟨*TuringMachine class methods*⟩+≡
```
1 public void setInitialState(State state) {
2     initialState = state;
3 }
```

Die Simulation der Turingmaschine kann mit der Methode run gestartet werden. Dabei werden zunächst bestimmte Variablen initialisiert und anschließend in einer Schleife die einzelnen Schritte der Turingmaschine abgearbeitet.

⟨*TuringMachine class methods*⟩+≡
```
1 public bool run(string input) {
2     ⟨Initialization run method 226⟩
3
4     while (true) {
5         ⟨Iteration run method 227⟩
6     }
7 }
```

Beim Start der Methode wird zunächst der aktuelle Zustand auf den Startzustand zurückgesetzt.

⟨*Initialization run method*⟩≡
```
1 current = initialState;
```

Das Band der Turingmaschine wird mit der Eingabe input initialisiert.

⟨*Initialization run method*⟩+≡
```
1 tape = input;
```

Dabei muss Sorge dafür getragen werden, dass das Band genügend Speicher für die bevorstehende Berechnung zur Verfügung stellt. Der Eingabestring muss insbesondere mindestens die Größe besitzen, die später an Zellen auf dem Band zum Lesen und Schreiben benötigt werden.

Im letzten Schritt der Initialisierung wird die maximale Bandlänge zwischengespeichert und geprüft, ob der initiale Zustand gültig ist.

⟨*Initialization run method*⟩+≡
```
1 tapeLength = input.Length();
2
3 // check if the start state is configured properly
4 if (current == null) die("Turing machine has no start state");
```

In der Iteration der Methode wird zunächst der aktuelle Zustand in der Variablen previousState abgespeichert. Im nächsten Schritt wird mithilfe der step-Methode der Folgezustand ermittelt. Die step-Methode kann hierbei auch den Wert null zurückgeben, falls kein Folgezustand definiert ist. Dies wird in der folgenden Anweisung überprüft. Sollte dieser Fall eintreten, wird die Ausführung der Turingmaschine beendet, da die Eingabe nicht durch die Turingmaschine akzeptiert wird. Andernfalls wird untersucht, ob der neue Zustand ein Endzustand ist. Sollte dieser Fall eintreten, wird die Simulation der Turingmaschine ebenfalls angehalten, da die Eingabe akzeptiert wurde.

⟨*Iteration run method*⟩≡
```
State previousState = current;
State state = step();

if(state == null) {
    print("Input rejected in state: " + previousState.id);
    return false;
}

if (state.accept) {
    print("Input accepted in state: " + state.id);
    return true;
}

debug("Moved to state: " + state.id);
```

Die step-Methode ist dafür verantwortlich, dass zum aktuellen Zustand und dem aktuellen Eingabesymbol die entsprechende Transition gefunden wird, die beschreibt, welches Zeichen auf das Band geschrieben werden soll, wie der Lese-Schreibkopf bewegt und in welchen Folgezustand gewechselt werden soll. Wird keine gültige Transition gefunden, wird der Wert null zurückgegeben.

⟨*TuringMachine class methods*⟩+≡
```
private State step() {
    int i = 0;
    char input = tape[head];
    State state = current;

    // look for a transition on the given input
    for (int i = 0; i < state.transitionCount; i++) {
        ⟨Iteration step method 227⟩
    }

    return null;
}
```

In der for-Schleife wird jede Transition, die dem aktuellen Zustand zugeordnet ist, besucht und überprüft, ob sie zum aktuellen Eingabesymbol passt.

⟨*Iteration step method*⟩≡
```
Transition trans = state.transitions[i];
if (trans == null) throw new Exception("Transition retrieval error");

// check if this is a transition in the given char input
if (trans.input == input) {
    ⟨Transition found step method 227⟩
}
```

Zunächst erfolgt eine Debugausgabe, die den Benutzer darüber informiert, dass eine entsprechede Transition gefunden wurde.

⟨*Transition found step method*⟩≡
```
debug("Found transition for input " + input);
```

Der Folgezustand wird ermittelt. Sollte dieser gleich null sein, wird eine Ausnahme geworfen.

⟨*Transition found step method*⟩+≡
```
State next = trans.next;
if (next == null) throw new Exception("Transitions to NULL state");
```

Im nächsten Schritt wird auf das Band geschrieben.

⟨*Transition found step method*⟩+≡
```
debug("Writing " + trans.write);
tape[head] = trans.write;
```

Im Anschluss daran wird der Lese-Schreibkopf bewegt und geprüft, ob es dabei nicht zu Speicherzugriffsverletzungen kommt.

⟨*Transition found step method*⟩+≡
```
switch(trans.move) {
   case eDirection.Left:
      if (head > 0) {
         head--;
         debug("Moved head left");
      }
      break;
   case eDirection.Right:
      if (head + 1 >= tapeLength) {
         die("Machine walked off tape on right side");
      }
      head++;
      debug("Moved head right");
      break;
   case eDirection.Hold:
      debug("Did not move head");
      break;
}
```

Im letzten Schritt wird der aktuelle Zustand mit dem ermittelten nächsten Zustand überschrieben und der gefundene nächste Zustand zurückgegeben.

⟨*Transition found step method*⟩+≡
```
// move the machine to the next state
debug("Setting current state");
current = next;

return next;
```

B.4.2 Beispiel: Zweierpotenzen

In diesem Abschnitt wird ein Beispiel einer konkreten Turingmaschine vorgestellt, die die Sprache $L = \{0^{2^n} | n > 0\}$ akzeptiert, wobei n eine natürliche Zahl ist. D. h. jede Folge von Nullen, deren Länge (Anzahl von Nullen) einer Zweierpotenz entspricht, wird von der entsprechenden Turingmaschine akzeptiert. Beispielsweise

gehört die Zeichenfolge (das Wort) “00000000” zur Sprache L, weil diese aus einer Folge von acht Nullen besteht und sich die Zahl Acht als Zweierpotenz darstellen lässt ($8 = 2^3$). Dagegen gehört das Wort “000” nicht zur Sprache, weil die Zahl drei nicht durch eine Zweierpotenz dargestellt werden kann.

Das Zustandsdiagramm einer Turingmaschine, welche die Sprache L akzeptiert, ist in Abbildung B.2 dargestellt.

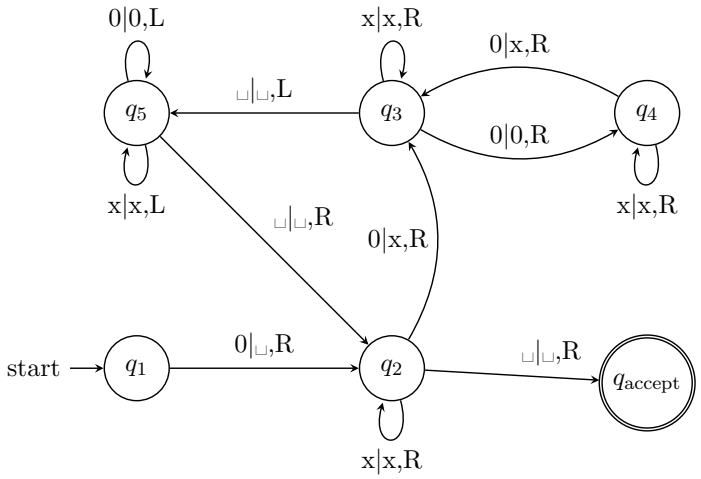


Abbildung B.2: Zustandsdiagramm einer Turingmaschine, die die Sprache $L = \{0^{2^n} | n > 0\}$ akzeptiert.

Die Umsetzung der Turingmaschine aus Abbildung B.2 mittels der Programmiersprache IFC-PL wird im Folgenden gezeigt. Die Turing Maschine wird auf Basis des vorher gezeigten Turingmaschinensimulators (siehe Abschnitt B.4.1) umgesetzt, der prinzipiell dazu genutzt werden kann, jede beliebige Turingmaschine zu simulieren. Das entsprechende simulierende IFC-PL-Programm ist im Standalone-Modus (siehe Abschnitt 5.3) implementiert und besitzt daher eine main-Funktion. Diese zeigt exemplarisch, wie die vorher definierten Klassen verwendet werden können, um eine Turingmaschine zu simulieren. Der Aufbau der main-Funktion gliedert sich wie folgt:

⟨*Main function*⟩≡

```
1 void main() {
2     TuringMachine machine = new TuringMachine();
3
4     ⟨State initialization main function 229⟩
5     ⟨Transition initialization main function 230⟩
6     ⟨Execute TM main function 230⟩
7 }
```

Für jeden Zustand Q wird entsprechend ein Zustandsobjekt erzeugt und entsprechend mit dem Konstruktor mitgeteilt, ob es sich beim aktuellen Zustand um einen akzeptierenden Endzustand handelt.

⟨*State initialization main function*⟩≡

```
1  State q1 = new State(false);
2  State q2 = new State(false);
3  State q3 = new State(false);
4  State q4 = new State(false);
5  State q5 = new State(false);
```

Nur der Zustand q_{accept} ist ein akzeptierender Endzustand. Der Startzustand ist der Zustand q_1.

⟨*State initialization main function*⟩+≡

```
1  State qaccept = new State(true);
2
3  machine.setInitialState(q1);
```

Nachfolgend werden alle Transitionen definiert. Eine Transition legt fest, welches Eingabesymbol erwartet wird, welches Bandsymbol geschrieben und in welche Richtung sich der Lese-Schreibkopf bewegen soll.

⟨*Transition initialization main function*⟩≡

```
1  Transition q1_q2_zero = new Transition('0', ' ', eDirection.Right, q2);
2  Transition q2_q2_x = new Transition('x', 'x', eDirection.Right, q2);
3  Transition q2_a_space = new Transition(' ', ' ', eDirection.Right,
     qaccept);
4  Transition q2_q3_zero = new Transition('0', 'x', eDirection.Right, q3);
5  Transition q3_q3_x = new Transition('x', 'x', eDirection.Right, q3);
6  Transition q3_q4_zero = new Transition('0', '0', eDirection.Right, q4);
7  Transition q3_q5_space = new Transition(' ', ' ', eDirection.Left, q5);
8  Transition q4_q3_zero = new Transition('0', 'x', eDirection.Right, q3);
9  Transition q4_q4_x = new Transition('x', 'x', eDirection.Right, q4);
10 Transition q5_q5_zero = new Transition('0', '0', eDirection.Left, q5);
11 Transition q5_q5_x = new Transition('x', 'x', eDirection.Left, q5);
12 Transition q5_q2_space = new Transition(' ', ' ', eDirection.Right, q2);
```

Im Anschluss werden den Zuständen entsprechend die Transitionen zugewiesen.

⟨*Transition initialization main function*⟩+≡

```
1  q1.add(q1_q2_zero);
2  q2.add(q2_q2_x);
3  q2.add(q2_a_space);
4  q2.add(q2_q3_zero);
5  q3.add(q3_q3_x);
6  q3.add(q3_q4_zero);
7  q3.add(q3_q5_space);
8  q4.add(q4_q3_zero);
9  q4.add(q4_q4_x);
10 q5.add(q5_q5_zero);
11 q5.add(q5_q5_x);
12 q5.add(q5_q2_space);
```

Nachdem alle Zustände und Übergänge definiert wurden, kann der Simulator ausgeführt werden.

⟨*Execute TM main function*⟩≡

```
1  string input = "0000 "; // two blanks are needed at the end of input
2
3  bool accepted = machine.run(input);
4
5  if(accepted) {
6      print("Accepted!");
7  }
8  else {
9      print("Rejected!");
10 }
```

Das gezeigte Programm liefert die Ausgabe Accepted!. Sie kann berechnen, ob eine Folge, bestehend aus Nullen, eine Zweierpotenz ist. Ähnlich können auch andere Turingmaschinen mit dem Simulator ausgeführt werden.

Anhang C

oipEXPRESS

C.1 Bison-Grammatik

Bison erwartet als Eingabe eine kontextfreie Grammatik und erzeugt, darauf basierend, einen Parser. Die Eingabegrammatik für Bison, die von oipExpress verwendet wird, ist im Folgenden dargestellt. Das Startsymbol der Grammatik lautet schema_decl.

```
1  schema_decl:
2     TOKEN_SCHEMA TOKEN_SIMPLE_ID TOKEN_SEMICOLON schema_body TOKEN_END_SCHEMA
3     TOKEN_SEMICOLON;
4
5  schema_body:
6     %empty
7   | type_decl schema_body
8   | entity_decl schema_body
9   | function_decl schema_body
10  | rule_decl schema_body;
11
12 function_decl:
13    function_head algorithm_head stmt function_decl1 TOKEN_END_FUNCTION
14    TOKEN_SEMICOLON;
15
16 function_decl1:
17    %empty
18  | stmt function_decl1;
19
20 function_head:
21    TOKEN_FUNCTION function_id function_head1 TOKEN_COLON parameter_type TOKEN_SEMICOLON;
22
23 function_head1:
24    %empty
25  | TOKEN_BRACKET_OPEN formal_parameter function_head2 TOKEN_BRACKET_CLOSE;
26
27 formal_parameter:
28    parameter_id formal_parameter1 TOKEN_COLON parameter_type;
29
30 formal_parameter1:
31    %empty
32  | TOKEN_COMMA parameter_id formal_parameter1;
33
34 function_head2:
35    %empty
36  | TOKEN_SEMICOLON formal_parameter function_head2;
37
38 algorithm_head:
39    algorithm_head1 algorithm_head2 algorithm_head3;
40
41 algorithm_head1:
42    %empty
43  | declaration algorithm_head1;
44
45 algorithm_head2:
46    %empty
47  | constant_decl;
48
49 algorithm_head3:
```

```
50      %empty
51    | local_decl;
52
53  local_decl:
54      TOKEN_LOCAL local_variable local_decl1 TOKEN_END_LOCAL TOKEN_SEMICOLON;
55
56  local_decl1:
57      %empty
58    | local_variable local_decl1;
59
60  local_variable:
61      variable_id local_variable1 TOKEN_COLON parameter_type local_variable2 TOKEN_SEMICOLON;
62
63  local_variable1:
64      %empty
65    | TOKEN_COMMA variable_id local_variable1;
66
67  local_variable2:
68      %empty
69    | TOKEN_ASSIGNMENT expression;
70
71  constant_decl:
72      TOKEN_CONSTANT constant_body constant_decl1 TOKEN_END_CONSTANT;
73
74  constant_decl1:
75      %empty
76    | constant_body constant_decl1;
77
78  constant_body:
79      constant_id TOKEN_COLON instantiable_type TOKEN_ASSIGNMENT expression TOKEN_SEMICOLON;
80
81  declaration :
82      entity_decl
83    | function_decl
84    | procedure_decl
85    | subtype_constraint_decl
86    | type_decl;
87
88  procedure_decl:
89      procedure_head algorithm_head procedure_decl1 TOKEN_END_PROCEDURE TOKEN_SEMICOLON;
90
91  procedure_decl1:
92      %empty
93    | stmt procedure_decl1;
94
95  procedure_head:
96      TOKEN_PROCEDURE procedure_id procedure_head1 TOKEN_SEMICOLON;
97
98  procedure_head1:
99      %empty
100   | TOKEN_BRACKET_OPEN procedure_head2 formal_parameter procedure_head3 TOKEN_BRACKET_CLOSE;
101
102 procedure_head2:
103     %empty
104   | TOKEN_VAR;
105
106 procedure_head3:
107     %empty
108   | TOKEN_SEMICOLON procedure_head2 formal_parameter procedure_head3;
109
110 procedure_id:
111     TOKEN_SIMPLE_ID;
112
113 stmt:
114     alias_stmt
115   | assignment_stmt
116   | case_stmt
117   | compound_stmt
118   | escape_stmt
119   | if_stmt
120   | null_stmt
121   | procedure_call_stmt
122   | repeat_stmt
123   | return_stmt
124   | skip_stmt;
125
126 alias_stmt:
127     TOKEN_ALIAS variable_id TOKEN_FOR general_ref alias_stmt1 TOKEN_SEMICOLON stmt alias_stmt2
128     TOKEN_END_ALIAS TOKEN_SEMICOLON;
129
130 alias_stmt1:
131     %empty
132   | qualifier alias_stmt1;
133
134  alias_stmt2:
135     %empty
136   | stmt alias_stmt2;
137
138 assignment_stmt:
139     general_ref assignment_stmt1 TOKEN_ASSIGNMENT expression TOKEN_SEMICOLON;
```

```
140
141  assignment_stmt1:
142     %empty
143    | qualifier assignment_stmt1;
144
145  case_stmt:
146     TOKEN_CASE selector TOKEN_OF case_stmt1 case_stmt2 TOKEN_END_CASE TOKEN_SEMICOLON;
147
148  case_stmt1:
149     %empty
150    | case_action case_stmt1;
151
152  case_stmt2:
153     %empty
154    | TOKEN_OTHERWISE TOKEN_COLON stmt;
155
156  case_action:
157     case_label case_action1 TOKEN_COLON stmt;
158
159  case_action1:
160     %empty
161    | TOKEN_COMMA case_label case_action1;
162
163  case_label:
164     expression;
165
166  procedure_ref:
167     procedure_id;
168
169  repeat_control:
170     repeat_control1 repeat_control2 repeat_control3;
171
172  repeat_control1:
173     %empty
174    | increment_control;
175
176  repeat_control2:
177     %empty
178    | while_control;
179
180  repeat_control3:
181     %empty
182    | until_control;
183
184  selector:
185     expression;
186
187  subtype_constraint_body:
188     subtype_constraint_body1 subtype_constraint_body2 subtype_constraint_body3;
189
190  subtype_constraint_body1:
191     %empty
192    | abstract_supertype;
193
194  subtype_constraint_body2:
195     %empty
196    | total_over;
197
198  subtype_constraint_body3:
199     %empty
200    | supertype_expression TOKEN_SEMICOLON;
201
202  subtype_constraint_head:
203     TOKEN_SUBTYPE_CONSTRAINT subtype_constraint_id TOKEN_FOR entity_ref TOKEN_SEMICOLON;
204
205  abstract_supertype:
206     TOKEN_ABSTRACT TOKEN_SUPERTYPE TOKEN_SEMICOLON;
207
208  compound_stmt:
209     TOKEN_BEGIN stmt compound_stmt1 TOKEN_END TOKEN_SEMICOLON;
210
211  compound_stmt1:
212     %empty
213    | stmt compound_stmt1;
214
215  escape_stmt:
216     TOKEN_ESCAPE TOKEN_SEMICOLON;
217
218  if_stmt:
219     TOKEN_IF logical_expression TOKEN_THEN stmt if_stmt1 if_stmt2 TOKEN_END_IF
220     TOKEN_SEMICOLON;
221
222  if_stmt1:
223     %empty
224    | stmt if_stmt1;
225
226  if_stmt2:
227     %empty
228    | TOKEN_ELSE stmt if_stmt1
229
```

```
230  null_stmt:
231      TOKEN_SEMICOLON;
232
233  procedure_call_stmt:
234      procedure_call_stmt1 procedure_call_stmt2 TOKEN_SEMICOLON;
235
236  procedure_call_stmt1:
237      built_in_procedure
238    | procedure_ref;
239
240  procedure_call_stmt2:
241      %empty
242    | actual_parameter_list;
243
244  repeat_stmt:
245      TOKEN_REPEAT repeat_control TOKEN_SEMICOLON stmt repeat_stmt1 TOKEN_END_REPEAT
246      TOKEN_SEMICOLON;
247
248  repeat_stmt1:
249      %empty
250    | stmt repeat_stmt1;
251
252  return_stmt:
253      TOKEN_RETURN return_stmt1 TOKEN_SEMICOLON;
254
255  return_stmt1:
256      %empty
257    | TOKEN_BRACKET_OPEN expression TOKEN_BRACKET_CLOSE;
258
259  skip_stmt:
260      TOKEN_SKIP TOKEN_SEMICOLON;
261
262  subtype_constraint_decl:
263      subtype_constraint_head subtype_constraint_body TOKEN_END_SUBTYPE_CONSTRAINT TOKEN_SEMICOLON;
264
265  built_in_procedure:
266      TOKEN_INSERT
267    | TOKEN_REMOVE;
268
269  increment_control:
270      variable_id TOKEN_ASSIGNMENT bound_1 TOKEN_TO bound_2 increment_control2;
271
272  increment_control2:
273      %empty
274    | TOKEN_BY increment;
275
276  subtype_constraint_id:
277      TOKEN_SIMPLE_ID;
278
279  total_over:
280      TOKEN_TOTAL_OVER TOKEN_BRACKET_OPEN entity_ref total_over1 TOKEN_BRACKET_CLOSE
281      TOKEN_SEMICOLON;
282
283  total_over1:
284      %empty
285    | TOKEN_COMMA entity_ref total_over1;
286
287  until_control:
288      TOKEN_UNTIL logical_expression;
289
290  while_control:
291      TOKEN_WHILE logical_expression;
292
293  bound_1:
294      numeric_expression;
295
296  bound_2:
297      numeric_expression;
298
299  increment:
300      numeric_expression;
301
302  type_decl:
303      TOKEN_TYPE TOKEN_SIMPLE_ID TOKEN_EQUAL underlying_type TOKEN_SEMICOLON type_decl2
304      TOKEN_END_TYPE TOKEN_SEMICOLON;
305
306  type_decl2:
307      %empty
308    | where_clause;
309
310  where_clause:
311      TOKEN_WHERE domain_rules;
312
313  domain_rules:
314          domain_rule TOKEN_SEMICOLON
315      |   domain_rules domain_rule TOKEN_SEMICOLON;
316
317  domain_rule:
318      rule_label_id TOKEN_COLON expression
319    | expression;
```

```
320
321  expression:
322      simple_expression expression1;
323
324  expression1:
325      %empty
326      | rel_op_extended simple_expression
327
328  simple_expression:
329      term simple_expression1;
330
331  simple_expression1:
332      %empty
333    | add_like_op term simple_expression1;
334
335  add_like_op:
336      TOKEN_PLUS
337    | TOKEN_MINUS
338    | TOKEN_OR
339    | TOKEN_XOR;
340
341  term:
342      factor term1;
343
344  term1:
345      %empty
346    | multiplication_like_op factor term1;
347
348  factor:
349      simple_factor factor1;
350
351  factor1:
352      %empty
353    | TOKEN_EXP simple_factor factor1;
354
355  simple_factor:
356      aggregate_initializer
357    | entity_constructor
358    | enumeration_reference
359    | interval
360    | query_expression
361    | primary
362    | unary_op primary
363    | TOKEN_BRACKET_OPEN expression TOKEN_BRACKET_CLOSE
364    | unary_op TOKEN_BRACKET_OPEN expression TOKEN_BRACKET_CLOSE;
365
366  query_expression:
367      TOKEN_QUERY TOKEN_BRACKET_OPEN variable_id TOKEN_ALL_IN aggregate_source TOKEN_STROKE
368      logical_expression TOKEN_BRACKET_CLOSE;
369
370  aggregate_source:
371      simple_expression;
372
373  logical_expression:
374      expression;
375
376  entity_constructor:
377      entity_ref TOKEN_BRACKET_OPEN entity_constructor1 TOKEN_BRACKET_CLOSE
378
379  entity_constructor1:
380      %empty
381    | expression
382    | expression TOKEN_COMMA entity_constructor1;
383
384  unary_op:
385      TOKEN_PLUS
386    | TOKEN_MINUS
387    | TOKEN_NOT;
388
389  interval:
390      TOKEN_CURLY_BRACKETS_OPEN interval_low interval_op interval_item interval_op interval_high
391      TOKEN_CURLY_BRACKETS_CLOSE;
392
393  interval_op:
394       TOKEN_LESS_THAN
395     | TOKEN_LESS_THAN_EQUAL;
396
397  interval_low:
398      simple_expression;
399
400  interval_item:
401      simple_expression;
402
403  interval_high:
404      simple_expression;
405
406  multiplication_like_op:
407      TOKEN_MULTIPLY
408    | TOKEN_SLASH
409    | TOKEN_DIV
```

```
410     | TOKEN_MOD
411     | TOKEN_AND
412     | TOKEN_CONCAT_OP;
413
414   rel_op:
415       TOKEN_LESS_THAN
416     | TOKEN_GREATER_THAN
417     | TOKEN_LESS_THAN_EQUAL
418     | TOKEN_GREATER_THAN_EQUAL
419     | TOKEN_NOT_EQUAL
420     | TOKEN_EQUAL
421     | TOKEN_INST_NOT_EQUAL
422     | TOKEN_INST_EQUAL;
423
424   rel_op_extended:
425       rel_op
426     | TOKEN_IN
427     | TOKEN_LIKE;
428
429   built_in_function:
430       TOKEN_ABS
431     | TOKEN_ACOS
432     | TOKEN_ASIN
433     | TOKEN_ATAN
434     | TOKEN_BLENGTH
435     | TOKEN_COS
436     | TOKEN_EXISTS
437     | TOKEN_EXP
438     | TOKEN_FORMAT
439     | TOKEN_HIBOUND
440     | TOKEN_HIINDEX
441     | TOKEN_LENGTH
442     | TOKEN_LOBOUND
443     | TOKEN_LOINDEX
444     | TOKEN_LOG
445     | TOKEN_LOG2
446     | TOKEN_LOG10
447     | TOKEN_NVL
448     | TOKEN_ODD
449     | TOKEN_ROLESOF
450     | TOKEN_SIN
451     | TOKEN_SIZEOF
452     | TOKEN_SQRT
453     | TOKEN_TAN
454     | TOKEN_TYPEOF
455     | TOKEN_USEDIN
456     | TOKEN_VALUE
457     | TOKEN_VALUE_IN
458     | TOKEN_VALUE_UNIQUE;
459
460   primary:
461       literal
462     | qualifiable_factor primary1;
463
464   primary1:
465       %empty
466     | qualifier primary1;
467
468   literal:
469       binary_literal
470     | logical_literal
471     | real_literal
472     | string_literal;
473
474   logical_literal:
475       TOKEN_FALSE
476     | TOKEN_TRUE
477     | TOKEN_UNKNOWN;
478
479   binary_literal:
480       TOKEN_BINARY_LITERAL;
481
482   string_literal:
483       simple_string_literal;
484
485   simple_string_literal:
486       TOKEN_SIMPLE_STRING_LITERAL;
487
488   real_literal:
489       integer_literal
490     | TOKEN_REAL_LITERAL;
491
492   integer_literal:
493       TOKEN_INTEGER_LITERAL;
494
495   qualifier:
496       attribute_qualifier
497     | group_qualifier
498     | index_qualifier;
499
```

```
500  index_qualifier:
501      TOKEN_SQUARE_BRACKET_OPEN index_1 TOKEN_SQUARE_BRACKET_CLOSE
502    | TOKEN_SQUARE_BRACKET_OPEN index_1 TOKEN_COLON index_2
503      TOKEN_SQUARE_BRACKET_CLOSE;
504
505  index_1:
506      index;
507
508  index_2:
509      index;
510
511  index:
512      numeric_expression;
513
514  qualifiable_factor:
515      attribute_ref
516    | constant_factor
517    | function_call
518    | general_ref
519    | population;
520
521  population:
522      entity_ref;
523
524  general_ref:
525      parameter_ref
526    | variable_ref
527
528  parameter_ref:
529      parameter_id;
530
531  parameter_id:
532      TOKEN_SIMPLE_ID;
533
534  variable_ref:
535      variable_id;
536
537  variable_id:
538      TOKEN_SIMPLE_ID;
539
540  function_call:
541      built_in_function function_call1
542    | function_ref function_call1;
543
544  function_call1:
545      %empty
546    | actual_parameter_list;
547
548  function_ref:
549      function_id;
550
551  function_id:
552      TOKEN_SIMPLE_ID;
553
554  actual_parameter_list:
555       TOKEN_BRACKET_OPEN actual_parameter_list1 TOKEN_BRACKET_CLOSE;
556
557  actual_parameter_list1:
558      %empty
559   | parameter
560   | parameter TOKEN_COMMA actual_parameter_list1;
561
562  parameter:
563      expression;
564
565  constant_factor:
566      built_in_constant
567    | constant_ref;
568
569  constant_ref:
570      constant_id;
571
572  constant_id:
573      TOKEN_SIMPLE_ID;
574
575  built_in_constant:
576      TOKEN_PI
577    | TOKEN_SELF
578    | TOKEN_QUESTION_MARK;
579
580  underlying_type:
581      concrete_types
582    | constructed_types;
583
584  constructed_types:
585      enumeration_type
586    | select_type;
587
588  select_type:
589      TOKEN_SELECT select_list;
```

```
590
591 select_list:
592     TOKEN_BRACKET_OPEN named_types select_list2 TOKEN_BRACKET_CLOSE;
593
594 select_list2:
595     %empty
596   | TOKEN_COMMA named_types select_list2;
597
598 named_types:
599     TOKEN_SIMPLE_ID;
600
601 enumeration_type:
602     TOKEN_ENUMERATION TOKEN_OF enumeration_items;
603
604 enumeration_items:
605     TOKEN_BRACKET_OPEN enumeration_id enumeration_items2 TOKEN_BRACKET_CLOSE;
606
607 enumeration_items2:
608     %empty
609   | TOKEN_COMMA enumeration_id enumeration_items2;
610
611 enumeration_id:
612     TOKEN_SIMPLE_ID;
613
614 aggregation_types:
615     array_type
616   | bag_type
617   | list_type
618   | set_type;
619
620 array_type:
621     TOKEN_ARRAY bound_spec TOKEN_OF array_type1 array_type2 instantiable_type;
622
623 array_type1:
624     %empty
625     | TOKEN_OPTIONAL;
626
627 array_type2:
628     %empty
629     | TOKEN_UNIQUE;
630
631 bag_type:
632     TOKEN_BAG bag_type1 TOKEN_OF instantiable_type;
633
634 bag_type1:
635     %empty
636   | bound_spec;
637
638 list_type:
639     TOKEN_LIST list_type1 TOKEN_OF list_type2 instantiable_type;
640
641 list_type1:
642     %empty
643   | bound_spec;
644
645 list_type2:
646     %empty
647     | TOKEN_UNIQUE;
648
649 set_type:
650     TOKEN_SET bound_spec TOKEN_OF instantiable_type;
651
652 concrete_types:
653     aggregation_types
654   | simple_types
655   | type_ref;
656
657 simple_types:
658     TOKEN_BINARY
659   | TOKEN_BINARY TOKEN_BRACKET_OPEN TOKEN_INTEGER_LITERAL TOKEN_BRACKET_CLOSE
660   | TOKEN_BOOLEAN
661   | TOKEN_INTEGER
662   | TOKEN_LOGICAL
663   | TOKEN_NUMBER
664   | TOKEN_REAL
665   | string_type;
666
667 string_type:
668     TOKEN_STRING
669   | TOKEN_STRING width_spec;
670
671 instantiable_type:
672     concrete_types
673   | entity_ref;
674
675 bound_spec:
676     TOKEN_SQUARE_BRACKET_OPEN lower_bound TOKEN_COLON upper_bound
677     TOKEN_SQUARE_BRACKET_CLOSE;
678
679 lower_bound:
```

```
680     numeric_expression;
681
682   upper_bound:
683     numeric_expression;
684
685   aggregate_initializer:
686       TOKEN_SQUARE_BRACKET_OPEN aggregate_initializer1 TOKEN_SQUARE_BRACKET_CLOSE;
687
688   aggregate_initializer1:
689     %empty
690     | element
691     | element TOKEN_COMMA aggregate_initializer1;
692
693   element:
694       expression
695     | expression TOKEN_COLON repetition;
696
697   repetition:
698       numeric_expression;
699
700   numeric_expression:
701       simple_expression;
702
703   width:
704       numeric_expression;
705
706   width_spec:
707       TOKEN_BRACKET_OPEN width TOKEN_BRACKET_CLOSE;
708     | TOKEN_BRACKET_OPEN width TOKEN_BRACKET_CLOSE TOKEN_FIXED;
709
710   entity_decl:
711       entity_head entity_body TOKEN_END_ENTITY TOKEN_SEMICOLON
712       {
713           oip::Schema::getInstance().addEntity(currentEntity);
714           currentEntity = Entity();
715           g_hasParent = false;
716
717           clear(attribute_types);
718       };
719
720   entity_head:
721       TOKEN_ENTITY entity_id TOKEN_SEMICOLON
722       {
723           std::string entityName = entity_ids.top();
724           entity_ids.pop();
725
726           currentEntity.setName(entityName);
727       }
728     | TOKEN_ENTITY entity_id subsuper TOKEN_SEMICOLON
729       {
730           std::string entityName = entity_ids.top();
731           entity_ids.pop();
732
733           currentEntity.setName(entityName);
734
735           if(g_hasParent)
736               currentEntity.setParentEntity(g_parentName);
737       };
738
739   entity_id:
740       TOKEN_SIMPLE_ID
741       {
742           entity_ids.push($1);
743       }
744
745   subsuper:
746       supertype_constraint
747     | subtype_declaration
748     | supertype_constraint subtype_declaration;
749
750   supertype_constraint:
751       supertype_rule
752     | abstract_supertype_declaration;
753
754   supertype_rule:
755       TOKEN_SUPERTYPE subtype_constraint;
756
757   subtype_constraint:
758       TOKEN_OF TOKEN_BRACKET_OPEN supertype_expression TOKEN_BRACKET_CLOSE;
759
760   supertype_expression:
761       supertype_factor
762     | one_of;
763
764   one_of:
765       TOKEN_ONEOF TOKEN_BRACKET_OPEN supertype_expression one_of2 TOKEN_BRACKET_CLOSE;
766
767   one_of2:
768       %empty
769     | TOKEN_COMMA supertype_expression one_of2;
```

```
770
771  supertype_factor:
772      supertype_term;
773
774  supertype_term:
775      entity_ref;
776
777  subtype_declaration:
778      TOKEN_SUBTYPE TOKEN_OF TOKEN_BRACKET_OPEN TOKEN_SIMPLE_ID
779      TOKEN_BRACKET_CLOSE;
780
781  entity_ref:
782      TOKEN_SIMPLE_ID;
783
784  entity_body:
785      %empty
786    | entity_body1 entity_body2 entity_body3 entity_body4 entity_body5;
787
788  entity_body1:
789      %empty
790    | explicit_attr entity_body1;
791
792  entity_body2:
793      %empty
794    | derive_clause;
795
796  entity_body3:
797      %empty
798    | inverse_clause;
799
800  entity_body4:
801      %empty
802    | unique_clause;
803
804  entity_body5:
805      %empty
806    | where_clause;
807
808  derive_clause:
809      TOKEN_DERIVE derived_attr derive_clause1;
810
811  derive_clause1:
812      %empty
813    | derived_attr derive_clause1;
814
815  derived_attr:
816      attribute_decl TOKEN_COLON parameter_type TOKEN_ASSIGNMENT expression TOKEN_SEMICOLON;
817
818  unique_clause:
819      TOKEN_UNIQUE unique_rule TOKEN_SEMICOLON unique_clause1;
820
821  unique_clause1:
822      %empty
823    | unique_rule TOKEN_SEMICOLON unique_clause1 ;
824
825  unique_rule:
826      rule_label_id TOKEN_COLON referenced_attribute unique_rule1
827    | referenced_attribute unique_rule1;
828
829  unique_rule1:
830      %empty
831    | TOKEN_COMMA referenced_attribute unique_rule1;
832
833  referenced_attribute:
834      attribute_ref
835    | qualified_attribute;
836
837  rule_label_id:
838      TOKEN_SIMPLE_ID;
839
840  inverse_clause:
841      TOKEN_INVERSE inverse_attr inverse_clause2;
842
843  inverse_clause2:
844      %empty
845    | inverse_attr inverse_clause2;
846
847  inverse_attr:
848      attribute_decl TOKEN_COLON inverse_attr1 entity_ref TOKEN_FOR attribute_ref inverse_attr4
849      TOKEN_SEMICOLON;
850
851  inverse_attr1:
852      %empty
853    | inverse_attr2 inverse_attr3 TOKEN_OF;
854
855  inverse_attr2:
856      TOKEN_SET
857    | TOKEN_BAG;
858
859  inverse_attr3:
```

```
860     %empty
861   | bound_spec;
862
863   inverse_attr4:
864     %empty
865   | entity_ref TOKEN_PERIOD;
866
867   attribute_ref:
868     attribute_id;
869
870   explicit_attr:
871      attribute_decl explicit_attr1 TOKEN_COLON TOKEN_OPTIONAL parameter_type TOKEN_SEMICOLON
872    | attribute_decl explicit_attr1 TOKEN_COLON parameter_type TOKEN_SEMICOLON;
873
874   explicit_attr1:
875     %empty
876   | TOKEN_COMMA attribute_decl;
877
878   parameter_type:
879     generalized_types
880   | named_types
881   | simple_types;
882
883   generalized_types:
884     aggregate_type
885   | general_aggregation_types
886   | generic_entity_type
887   | generic_type;
888
889   aggregate_type:
890     TOKEN_AGGREGATE TOKEN_OF parameter_type
891   | TOKEN_AGGREGATE TOKEN_COLON type_label TOKEN_OF parameter_type;
892
893   generic_entity_type:
894     TOKEN_GENERIC_ENTITY
895   | TOKEN_GENERIC_ENTITY TOKEN_COLON type_label;
896
897   generic_type:
898     TOKEN_GENERIC TOKEN_COLON type_label
899   | TOKEN_GENERIC;
900
901   type_label:
902     type_label_id;
903
904   type_label_id:
905     TOKEN_SIMPLE_ID;
906
907   general_aggregation_types:
908     general_array_type
909   | general_bag_type
910   | general_list_type
911   | general_set_type;
912
913   general_array_type:
914     TOKEN_ARRAY general_array_type1 TOKEN_OF general_array_type2 general_array_type3 parameter_type;
915
916   general_array_type1:
917     %empty
918   | bound_spec;
919
920   general_array_type2:
921     %empty
922   | TOKEN_OPTIONAL;
923
924   general_array_type3:
925     %empty
926   | TOKEN_UNIQUE;
927
928   general_bag_type:
929     TOKEN_BAG TOKEN_OF parameter_type
930   | TOKEN_BAG bound_spec TOKEN_OF parameter_type;
931
932   general_list_type:
933     TOKEN_LIST general_list_type1 TOKEN_OF general_list_type2 parameter_type;
934
935   general_list_type1:
936     %empty
937   | bound_spec;
938
939   general_list_type2:
940     %empty
941   | TOKEN_UNIQUE;
942
943   general_set_type:
944     TOKEN_SET TOKEN_OF parameter_type
945   | TOKEN_SET bound_spec TOKEN_OF parameter_type;
946
947   attribute_decl:
948     attribute_id;
949   | redeclared_attribute;
```

```
950
951  redeclared_attribute:
952      qualified_attribute;
953
954  qualified_attribute:
955      TOKEN_SELF group_qualifier attribute_qualifier;
956
957  group_qualifier:
958      TOKEN_BACKSLASH entity_ref;
959
960  attribute_qualifier:
961      TOKEN_PERIOD attribute_ref;
962
963  attribute_id:
964      TOKEN_SIMPLE_ID;
965
966  enumeration_reference:
967      enumeration_reference1 enumeration_ref;
968
969  enumeration_reference1:
970      %empty
971    | type_ref TOKEN_PERIOD;
972
973  enumeration_ref:
974      enumeration_id;
975
976  type_ref:
977      type_id;
978
979  type_id:
980      TOKEN_SIMPLE_ID;
981
982  abstract_supertype_declaration:
983      TOKEN_ABSTRACT TOKEN_SUPERTYPE subtype_constraint
984    | TOKEN_ABSTRACT TOKEN_SUPERTYPE;
985
986  subtype_constraint:
987      TOKEN_OF TOKEN_BRACKET_OPEN supertype_expression TOKEN_BRACKET_CLOSE;
988
989  supertype_expression:
990      supertype_factor;
991
992  supertype_factor:
993      supertype_term;
994
995  supertype_term:
996      entity_ref
997    | one_of
998    | TOKEN_BRACKET_OPEN supertype_expression TOKEN_BRACKET_CLOSE;
999
1000 rule_decl:
1001     rule_head algorithm_head rule_decl1 where_clause TOKEN_END_RULE TOKEN_SEMICOLON;
1002
1003 rule_decl1:
1004     %empty
1005   | stmt rule_decl1;
1006
1007 rule_head:
1008     TOKEN_RULE rule_id TOKEN_FOR TOKEN_BRACKET_OPEN entity_ref rule_head1
1009     TOKEN_BRACKET_CLOSE TOKEN_SEMICOLON;
1010
1011 rule_head1:
1012     %empty
1013   | TOKEN_COMMA entity_ref rule_head1;
1014
1015 rule_id:
1016     TOKEN_SIMPLE_ID;
```

Literaturverzeichnis

Aho, A. V., Lam, M. S., Sethi, R. & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Albahari, J. & Albahari, B. (2015). *C# 6.0 in a Nutshell: The Definitive Reference* (6th Aufl.). O'Reilly Media, Inc.

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Amann, J. (2015). Einbettung von prozeduralem Wissen in Produktdatenmodelle am Beispiel der Richtlinien für die Anlage von Landstraßen mithilfe der Programmiersprache IFCPL. In: *Proceedings of the 27th Forum Bauinformatik*, Aachen, Germany.

Amann, J. & Borrmann, A. (2015a). Creating a 3D-BIM-compliant road design based on IFC alignment originating from an OKSTRA-accordant 2D road design using the TUM Open Infra Platform and the OKSTRA class library. Forschungsbericht, Technische Universität München.

Amann, J. & Borrmann, A. (2015b). Creating a 3D-BIM-compliant road design based on IFC alignment originating from an OKSTRA-accordant 2D road design using the TUM Open Infra Platform and the OKSTRA class library. Forschungsbericht, Technische Universität München.

Amann, J. & Borrmann, A. (2015c). Open BIM for Infrastructure – mit OKSTRA und IFC Alignment zur internationalen Standardisierung des Datenaustauschs. In: *Tagungsband zum 6. OKSTRA-Symposium*, Köln, Deutschland.

Amann, J. & Borrmann, A. (2016). An XML-based approach for the exchange of application defined algorithms in building information models. In: *Proc. of the 33rd CIB W78 Conference 2016*, Brisbane, Australia.

Amann, J., Borrmann, A., Chipman, T., Lebegue, E., Liebich, T. & Scarponcini, P. (2014). P6 IFC Alignment – Conceptual Model. (1), S. 1–22.

Amann, J., Borrmann, A., Hegemann, F., Jubierre, J., Flurl, M., Koch, C. & König, M. (2013). A Refined Product Model for Shield Tunnels Based on a

Generalized Approach for Alignment Representation. In: *Proc. of the ICCBEI 2013*, Tokyo, Japan.

Amann, J., Schöttl, F., Singer, D., Kern, M., Widner, A., Geisler, P., Below, D., Hecht, H., Gupta, N., Mustafa, A., Markič, v. & Borrmann, A. (2016). TUM Open Infra Platform 2018.

Amann, J., Singer, D. & Borrmann, A. (2015). Extension of the upcoming IFC alignment standard with cross sections for road design. In: *Proc. of the ICCBEI 2015*, Tokyo, Japan.

Amann, J., Tauscher, E. & Borrmann, A. (2015). BIM-Programmierwerkzeuge. In: A. Borrmann, M. König, C. Koch, & J. Beetz (Hrsg.), *Building Information Modeling*. Springer Fachmedien Wiesbaden.

Amann, J., Weber, B. & Wüthrich, C. A. (2013). Using Image Quality Assessment to Test Rendering Algorithms. In: *21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in co-operation with EUROGRAPHICS Association, WSCG 2013, Plzen, Czech Republic, June 24-27, 2013*, S. 205–214.

Amor, R. (2015). Analysis of the Evolving IFC Schema. In: *Proceedings of the CIB W78 conference 2015*, Eindhoven, Netherlands.

Amor, R., Jiang, Y. & Chen, X. (2007). BIM in 2007–are we there yet. *… of CIB W78 conference on Bringing …*.

Anderl, R. (2000). *STEP : standard for the exchange of product model data ; eine Einführung in die Entwicklung, Implementierung und industrielle Nutzung der Normenreihe ISO 10303 (STEP)*. Stuttgart; Leipzig: Teubner.

Appel, A. W. (1997). *Modern Compiler Implementation in C: Basic Techniques*. New York, NY, USA: Cambridge University Press.

Autodesk (2018). AutoCAD Civil 3D. `https://www.autodesk.de/products/autocad-civil-3d/overview`. Letzter Zugriff am 8. April 2018.

BMVI (2015). *Stufenplan Digitales Planen und Bauen*. Bundesministerium für Verkehr und digitale Infrastruktur.

Borrmann, A., Amann, J., Chipman, T., Hyvärinen, J., Liebich, T., Muhic, S., Mol, L., Plume, J. & Scarponcini, P. (2017). IFC Infra Overall Architecture Project Documentation and Guidelines. *buildingSMART*.

Borrmann, A., Beetz, J., Koch, C. & Liebich, T. (2015). *Industry Foundation Classes – Ein herstellerunabhängiges Datenmodell für den gesamten Lebenszyklus eines Bauwerks*, S. 83–127. Wiesbaden: Springer Fachmedien Wiesbaden.

Borrmann, A., König, M., Koch, C. & Beetz, J. (2015). *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. VDI-Buch. Springer Fachmedien Wiesbaden.

Borrmann, A., König, M., Koch, C. & Beetz, J. (2015). *Building Information Modeling: Technologische Grundlagen und industrielle Praxis*. Springer Vieweg.

Braun, A., Tuttas, S., Borrmann, A. & Stilla, U. (2014). Towards automated construction progress monitoring using BIM-based point cloud processing. In: *eWork and eBusiness in Architecture, Engineering and Construction: ECPPM 2014*, Vienna, Austria.

Brundage, M. (2004). *XQuery: The XML Query Language*. Pearson Higher Education.

Bühlmann, T. (1996). *Parametrik in der Produktdatenmodellierung*. Dissertation, ETH Zurich.

buildingSMART (2007a). IFC Parametric Schema. `http://www.buildingsmart-tech.org/future/old/ifc-future-extensions/project-proposals/pa-1-parametric/IfcParametric2x3.exp/view`. Letzter Zugriff am 7. Juli 2018.

buildingSMART (2007b). PA-1 Parametric documentation. `http://www.buildingsmart-tech.org/future/old/ifc-future-extensions/project-proposals/pa-1-parametric/pa-1-parametric-documentation`. Letzter Zugriff am 13. Mai 2018.

buildingSMART (2014). IFC Alignment Schema.

buildingSMART (2018a). Chapter Directory - buildingSMART. `https://www.buildingsmart.org/chapters/chapter-directory/`. Letzter Zugriff am 3. April 2018.

buildingSMART (2018b). IFC 2.3 Documentation. `http://www.buildingsmart-tech.org/ifc/IFC2x3/TC1/html/`. Letzter Zugriff am 8. April 2018.

buildingSMART (2018c). IFC 4.1 Documentation. `http://www.buildingsmart-tech.org/ifc/IFC4x1/final/html/`. Letzter Zugriff am 8. April 2018.

buildingSMART (2018d). IFC4x1 Alignment Extension - 1.0.1. `http://www.buildingsmart-tech.org/mvd/IFC4x1/Alignment/1.0.1/html/`. Letzter Zugriff am 10. Mai 2018.

buildingSmart Model-Support Group (2015). IFC Alignment Candidate Standard.

Bundesanstalt für Straßenwesen (2018). www.okstra.de. Letzter Zugriff am 26. Juni 2018.

Bundesministerium für Verkehr, Bau- und Wohnungswesen (2010). Allgemeines Rundschreiben Straßenbau Nr. 12/2000 Sachgebiet 19.2: Straßeninformationsbanken. Forschungsbericht 12.

Bungartz, H., Griebel, M. & Zenger, C. (2013). *Einführung in die Computergraphik: Grundlagen, Geometrische Modellierung, Algorithmen*. Mathematische Grundlagen der Informatik. Vieweg+Teubner Verlag.

Chaplier, J., That, T. N., Hewatt, M., Gallée, G., Sa, O., Unies, N. & France, M. (2012). Toward a standard : RoadXML , the road network database format. *Driving Simulation Conference 2010 Europe*, S. 211–220.

CMS (2018a). IFC-PL. `https://bitbucket.org/tumcms/ifcpl-public`. Lehrstuhl für Computergestützte Modellierung und Simulation (CMS), Technische Universität München, Letzter Zugriff am 30. April 2018.

CMS (2018b). TUM Open Infra Platform. `https://www.cms.bgu.tum.de/oip`. Lehrstuhl für Computergestützte Modellierung und Simulation (CMS), Technische Universität München, Letzter Zugriff am 8. April 2018.

CMS (2018c). TUM Open Infra Platform 2018. `https://www.cms.bgu.tum.de/oip`. Lehrstuhl für Computergestützte Modellierung und Simulation (CMS), Technische Universität München, Letzter Zugriff am 02. Mai 2018.

CMS (2018d). TUM Open Infra Platform Early Binding EXPRESS Generator. `https://bitbucket.org/tumcms/oipexpress`. Lehrstuhl für Computergestützte Modellierung und Simulation (CMS), Technische Universität München, Letzter Zugriff am 17. April 2018.

Dang, B., Gazet, A., Bachaalany, E. & Josse, S. (2014). *Practical Reverse Engineering: X86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation* (1st Aufl.). Wiley Publishing.

Daum, S. & Borrmann, A. (2015). Simplifying the Analysis of Building Information Models Using tQL4BIM and vQL4BIM. In: *Proc. of the EG-ICE 2015*, Eindhoven, The Netherlands.

Daum, S., Borrmann, A. & Kolbe, T. H. (2016). A Spatio-Semantic Query Language for the Integrated Analysis of City Models and Building Information Models. In: A. Abdul-Rahman (Hrsg.), *Advances in 3D Geoinformation*. Springer International Publishing.

Dimyadi, J. & Amor, R. (2013). Automated Building Code Compliance Checking – Where is it at? In: *Proceedings of CIB WBC 2013*, Brisbane, Australia, S. 172–185.

DuCharme, B. (2013). *Learning SPARQL: Querying and Updating with SPARQL 1.1*. O'Reilly Media.

Ducloux, P. & Millet, G. (2009). Road Network Description XML Format Specification. S. 1–88.

Eastman, C., Lee, J. M., Jeong, Y. S. & Lee, J. K. (2009). Automatic rule-based checking of building designs. *Automation in Construction* 18(8), S. 1011–1033.

Eastman, C., Teicholz, P., Sacks, R. & Liston, K. (2008). *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*. Wiley Publishing.

ECMA International (2006, Juli). *Standard ECMA-334 - C# Language Specification* (4 Aufl.).

Feser, B. & Rosenthal, R. (2004). *Entwicklung des Objektes "Dynamisches Querprofil"*. Bremerhaven: Wirtschaftsverl. NW, Verl. fr Neue Wiss.

FGSV (2012). Richtlinien für die Anlage von Landstraßen.

Flurl, M. H. J. (2016). *Kollaborative Modellierung und simulationsgestützte Evaluierung trassenbasierter Infrastrukturbauwerke*. Dissertation, Technische Universität München, München.

Free Software Foundation (2007). GNU General Public License, version 3. `http://www.gnu.org/licenses/gpl.html`. Letzter Zugriff am 17. April 2018.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Geißler, K. (2014). *Brückenentwurf*, S. 1–165. Wiley-VCH Verlag GmbH.

Gosling, J., Joy, B., Steele, G. L., Bracha, G. & Buckley, A. (2014). *The Java Language Specification, Java SE 8 Edition* (1st Aufl.). Addison-Wesley Professional.

Graphisoft (2018). ArchiCAD GDL Referenzhandbuch. `https://helpcenter.graphisoft.de/handbuecher/handbucher-zu-archicad-17/archicad-gdl-referenzhandbuch/`. Letzter Zugriff am 13. Mai 2018.

Hausknecht, K. & Liebich, T. (2016). *BIM-Kompendium.: Building Information Modeling als neue Planungsmethode*. Fraunhofer Irb Stuttgart.

Hegemann, F., Lehner, K. & König, M. (2012, 08). IFC-based product modeling for tunnel boring machines.

Hennecke, F., Müller, G., Werner, H., Möbius, G., Möserand, M. & Potthoff, H. (2000). *Handbuch Ingenieurvermessung, Bd.7, Verkehrsbau, Straßenbau*. Berlin: Verl. für Bauwesen.

Hopcroft, J. E. (2007). *Introduction to Automata Theory, Languages, and Computation* (3rd Aufl.). Pearson Addison Wesley.

Hyvärinen, J. (2011). Finnish InfraBIM Activities & Finnish Inframodel. *Powerpoint Präsentation*.

Illik, J. (2009). *Formale Methoden der Informatik: von der Automatentheorie zu Algorithmen und Datenstrukturen*. Reihe Technik. Expert-Verlag.

Intel Corporation (2011, Mai). *Intel 64 and IA-32 Architectures Software Developer's Manual: Instruction Set Reference, A-Z.*

ISO (2014, Dezember). *ISO/IEC 14882:2014 Information technology – Programming languages – C++.* Geneva, Switzerland: International Organization for Standardization.

ISO 10303-11:2004 (2014, November). Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual. Standard.

ISO 10303-21:2016-03 (2016, März). Industrial automation systems and integration - Product data representation and exchange - Part 21: Implementation methods: Clear text encoding of the exchange structure. Standard.

Jeong, S., Hou, R., Lynch, J. P., Sohn, H. & Law, K. H. (2017, Dezember). An Information Modeling Framework for Bridge Monitoring. *Adv. Eng. Softw.* 114(C), S. 11–31.

Ji, Y., Borrmann, A., Beetz, J. & Obergrießer, M. (2013). Exchange of parametric bridge models using a neutral data format. *Journal of computing in civil engineering* 27(6), S. 593–606.

Ji, Y., Borrmann, A., Beetz, J. & Obergrießer, M. (2013). Exchange of Parametric Bridge Models Using a Neutral Data Format. *Journal of Computing in Civil Engineering* 27(6), S. 593–606.

Jubierre, J. & Borrmann, A. (2014). A multi-scale product model for shield tunnels based on the Industry Foundation Classes. Forschungsbericht, Technische Universität München.

Kessenich, J., Sellers, G. & Shreiner, D. (2016). *OpenGL®Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V* (9 Aufl.). Addison-Wesley Professional.

Khan, A., Mordatch, I., Fitzmaurice, G., Matejka, J. & Kurtenbach, G. (2008). ViewCube: A 3D Orientation Indicator and Controller. In: *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, New York, NY, USA, S. 17–25. ACM.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal* 27(2), S. 97–111.

Kornbichler, D. (2000). Zwischenbericht zur Geometrischen Modellierung. S. 1–19.

Kornbichler, D. F. (1999). *Abbildung von Querprofilen im OKSTRA.* Bonn: Bundesministerium für Verkehr, Bau- und Wohnungswesen, Abteilung Strassenbau, Strassenverkehr.

Kozen, D. C. (1997). *Automata and Computability* (1st Aufl.). Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Krepp, S., Jahr, K., Bigontina, S., Bügler, M. & Borrmann, A. (2016). BIMsite - Towards a BIM-based Generation and Evaluation of Realization Variants Comprising Construction Methods, Site Layouts and Schedules. In: *Proc. of the EG-ICE Workshop on Intelligent Computing in Engineering*, Krakow, Poland.

Kuloyants, V. (2014). Entwicklung eines IFC-basierenden Datenaustauschstandards für den Unterbau von Brückenbauwerken. Diplomarbeit, Technische Universität München.

Laakso, M. & Kiviniemi, A. (2012). The IFC Standard - A Review of History, Development and Standardization. *ITcon Journal of Information Technology in Construction* 17, S. 134–161.

LandXML.org (2018). LandXML.org.

Lattner, C. & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, Washington, DC, USA, S. 75–. IEEE Computer Society.

Lebegue, E., Gual, J., Arthaud, G., Liebich, T., French, I. a. I., Eric, C., Cstb, I.-b. T. L., Gual, J., Setra, I.-b. P. L. & Arthaud, G. (2007). IFC-BRIDGE V2 Data Model. (November), S. 42.

Lee, G., Sacks, R. & Eastman, C. M. (2006). Specifying parametric building object behavior (BOB) for a building information modeling system. *Automation in Construction* 15(6), S. 758–776.

Lee, J.-K., Eastman, C. & Lee, Y. (2014). Implementation of a BIM Domain-specific Language for the Building Environment Rule and Analysis.

Leitzke, C. (2017). Gut aufgestellt – CARD/1 für Bahnvermesser. *interAktiv 1/2017*, S. 12–14.

Lenz, B., Nobis, C., Köhler, K., Mehlin, M., Follmer, R., D., G., Jesske, B. & Quandt, S. (2010). Mobilität in Deutschland 2008. Kurzbericht Struktur - Aufkommen - Emissionen - Trends.

Levine, J. R. (2009). *flex and bison - Unix text processing tools*. O'Reilly.

Liang, S. (1999). *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Java series. Addison-Wesley.

Lin, T. A. (1995). Building Smart - A Strategy for Implementing BIM Solution in Singapore. *Synthesis Journal 2006* 5, S. 117–124.

Luna, F. (2016). *Introduction to 3D Game Programming with DirectX 12*. USA: Mercury Learning & Information.

Martin, K. & Hoffman, B. (2008). *Mastering CMake 4th Edition* (4th Aufl.). USA: Kitware, Inc.

Martin, R. C. (1998). Java Gems. Chapter Java and C++: A Critical Comparison, S. 51–68. New York, NY, USA: Cambridge University Press.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Boston, MA: Prentice Hall.

Mazairac, W. & Beetz, J. (2013, Oktober). BIMQL - An Open Query Language for Building Information Models. *Adv. Eng. Inform.* 27(4), S. 444–456.

McConnell, S. (2004). *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press.

Mitchell, J. C. & Apt, K. (2001). *Concepts in Programming Languages*. New York, NY, USA: Cambridge University Press.

Nicholson-Cole, D. (2001). *The GDL cookbook 3 : the source of all that is good in GDL, and ArchiCAD tips and tricks*. Nottingham: Marmalade Graphics.

Nisbet, N. & Liebich, T. (2005). ifcXML implementation guide. *International Alliance for Interoperability*.

Orwell (2018). Dev-C++. `http://orwelldevcpp.blogspot.de/`.

Pauwels, P., Törmä, S., Beetz, J., Weise, M. & Liebich, T. (2015, 09). Linked Data in Architecture and Construction. 57, S. 175–177.

Pavel, R., Yakov, G. & Novgorodov, S. (2016). http://utf8everywhere.org/.

Pierce, B. C. (2002). *Types and Programming Languages* (1st Aufl.). The MIT Press.

Preidel, C. & Borrmann, A. (2015). Automated Code Compliance Checking Based on a Visual Language and Building Information Modeling. In: *Proc. of the 32nd ISARC 2015*, Oulu, Finland.

Preidel, C. & Borrmann, A. (2016). Towards code compliance checking on the basis of a visual programming language. *ITcon* 21, S. 402–421.

Preidel, C., Borrmann, A. & Beetz, J. (2015). BIM-gestützte Prüfung von Normen und Richtlinien. In: A. Borrmann, M. König, C. Koch, & J. Beetz (Hrsg.), *Building Information Modeling*. Springer Fachmedien Wiesbaden.

Rebolj, D., Tibaut, A., Čuš Babič, N., Magdič, A. & Podbreznik, P. (2008). Development and application of a road product model. *Automation in Construction* 17(6), S. 719 – 728.

Reis, G. D. & Hall, M. (2014). A Module System for C++. *Iso-N4047*, S. 1–22.

Richter, T. (2016). *Planung von Autobahnen und Landstrassen.* Wiesbaden: Springer Fachmedien Wiesbaden GmbH.

Scarponcini, P. (2006). TransXML: TransXML: Establishing standards for transportation data exchange. In: *Proc. of the CIB-W78 2006*.

Scarponcini, P. (2013). InfraGML Proposal (13-121) - OGC Land and Infrastructure DWG/SWG.

Schenck, D. A. & Wilson, P. R. (1994). *Information Modeling: The EXPRESS Way*. New York, NY, USA: Oxford University Press, Inc.

Schoenberg, I. J. (1964). SPLINE FUNCTIONS AND THE PROBLEM OF GRADUATION. *Proceedings of the National Academy of Sciences* 52(4), S. 947–950.

Scott, M. L. (2005). *Programming Language Pragmatics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Seidl, H., Wilhelm, R. & Hack, S. (2009). *Analyse und Transformation*. Number Bd. 3 in EXamen. press Series. Springer Berlin Heidelberg.

Sellers, G. & Kessenich, J. (2016). *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Always learning. Addison Wesley.

Shoemake, K. (1992). ARCBALL: A User Interface for Specifying Three-dimensional Orientation Using a Mouse. In: *Proceedings of the Conference on Graphics Interface '92*, San Francisco, CA, USA, S. 151–156. Morgan Kaufmann Publishers Inc.

Simpson, J. E. (2002). *XPath and XPointer: Locating Content in XML Documents*. Beijing: O'Reilly.

Singer, D. (2014). Entwicklung eines Prototyps für den Einsatz von Knowledge-based Engineering in frühen Phasen des Brückenentwurfs. Diplomarbeit, Technische Universität München.

Smith, J. & Nair, R. (2005). *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Socher, R. (2008). *Theoretische Grundlagen der Informatik: mit 31 Tabellen, 36 Beispielen und 75 Aufgaben mit Lösungen*. Lehrbücher zur Informatik. Hanser.

Thomsik, D. (2013). Objektkatalog für das Straßen- und Verkehrswesen Änderungsantrag A0042. Forschungsbericht, OKSTRA – Koordierungsstelle.

TIOBE-Software-BV (2018a, März). TIOBE Software: Tiobe Index. https://www.tiobe.com/tiobe-index/.

TIOBE-Software-BV (2018b, März). TIOBE Software: Tiobe Index. https://www.tiobe.com/tiobe-index/programming-languages-definition/.

Troelsen, A. (2003). *C# and the .Net Platform* (2 Aufl.). APress L. P.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42(1), S. 230–265.

van Berlo, L. (2014). BIM Service interface exchange (BIMSie). `https://www.nibs.org/?page=bsa_bimsie`. Letzter Zugriff am 11. Mai 2018.

Vandevoorde, D. & Josuttis, N. M. (2002, November). *C++ Templates: The Complete Guide* (1 Aufl.). Addison-Wesley Professional.

Veenhuis, W. (2017). Das Freie GAEB Buch. S. 1–62.

Vilgertshofer, S., Jubierre, J. & A., B. (2016). IfcTunnel - A proposal for a multi-scale extension of the IFC data model for shield tunnels under consideration of downward compatibility aspects. In: *11th European Conference on Product and Process Modelling*, Limassol, Cyprus.

Vince, J. (2011). *Quaternions for Computer Graphics* (1st Aufl.). Springer Publishing Company, Incorporated.

Vossen, G. & Witt, K.-U. (2006). *Grundkurs Theoretische Informatik: Eine anwendungsbezogene Einführung. Für Studierende in allen Informatik-Studiengängen.* Vieweg.

W3C (2014). XQuery 3.0: An XML Query Language. `https://www.w3.org/TR/xquery-30/`. Letzter Zugriff am 11. Mai 2018.

Walmsley, P. (2015). *XQuery: Search Across a Variety of XML Data.* O'Reilly Media.

Wijnholts, L. (2016). Automated Geometry Checking for Infrastructure Projects. Diplomarbeit, Eindhoven University of Technology.

Wilhelm, R., Seidl, H. & Hack, S. (2012). *Übersetzerbau: Band 2: Syntaktische und semantische Analyse.* eXamen.press. Springer Berlin Heidelberg.

Wirth, N. (2011). *Grundlagen und Techniken des Compilerbaus.* Oldenbourg.

Wood, D., Zaidman, M., Ruth, L. & Hausenblas, M. (2014). *Linked Data* (1st Aufl.). Greenwich, CT, USA: Manning Publications Co.

Xu, R. X. R., Solihin, W. & Huang, Z. H. Z. (2004). Code Checking and Visualization of an Architecture Design. *IEEE Visualization 2004*, S. 10p–10p.

Yabuki, N. (2009). Representation of caves in a shield tunnel product model. *Ework and Ebusiness in Architecture, Engineering and Construction*, S. 545–550.

Yabuki, N., Lebegue, E., Gual, J., Shitani, T. & Li, Z. (2006, 01). International Collaboration for Developing the Bridge Product Model "IFC-Bridge".

Yabuki, N., Lebegue, E., Gual, J., Shitani, T. & Zhantao, L. (2006). International Collaboration for Developing the Bridge Product Model IFC-Bridge. In: *Proceedings of the 2006 Joint International Conference on Computing and Decision Making in Civil and Building Engineering*, S. 1927–1936.

Yurchyshyna, A. & Zarli, A. (2009). An ontology-based approach for formalisation and semantic organisation of conformance requirements in construction. *Automation in Construction* 18(8), S. 1084–1098.

Zeiss, G. (2014). BIMSie: a standard API for BIM web services in the cloud. `http://geospatial.blogs.com/geospatial/2014/03/bimsie-a-standard-api-for-bim-web-services-in-the-cloud.html`. Letzter Zugriff am 11. Mai 2018.

Zelle, J. (2010). *Python Programming: An Introduction to Computer Science 2nd Edition*. Wilsonville, OR, USA: Franklin, Beedle & Associates Inc.

Ziering, E., Harrison, F., Board, N. R. C. U. T. R., of State Highway, A. A., Officials, T. & Program, N. C. H. R. (2007). *TransXML: XML Schemas for Exchange of Transportation Data*. NCHRP report. Transportation Research Board.

Zink, J., Pettineo, M. & Hoxley, J. (2011). *Practical Rendering and Computation with Direct3D 11* (1st Aufl.). Natick, MA, USA: A. K. Peters, Ltd.

Zwecker, E. (2014). Datenformate im Tief-, Erd- und Straßenbau. `https://doku.mts-online.de:8091/pages/viewpage.action?pageId=13467690`. Letzter Zugriff am 2. April 2018.